DOCKER PROJECTS HANDBOOK

Build, Deploy, and Secure Production-Ready Applications with Dockerfiles, Docker Compose, Networking, Volumes, and CI/CD



AUREN TEKLOR

AUREN TEKLOR

DOCKER PROJECTS HANDBOOK

BUILD • DEPLOY • SECURE • OPERATE

Build, Deploy, and Secure Production-Ready Applications with Dockerfiles, Docker Compose, Networking, Volumes, and CI/CD

Copyright © 2026 Auren Teklor. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, except as permitted by applicable law or by written permission of the rights holder. Short excerpts may be quoted in reviews or scholarly discussion where lawful.

Educational and software use

The code, configuration, shell commands, and lab exercises in this book are provided for learning. Readers may type, run, adapt, and test the book's original example code for personal and organizational learning and for applications they create; this permission does not extend to republishing the book or distributing its text and illustrations as a substitute for the book. Third-party packages, images, and example dependencies remain subject to their own licenses.

Technology changes. Commands, defaults, interface labels, security recommendations, service limits, and prices can differ by date, architecture, operating system, or vendor. Verify consequential actions against current official documentation and test in a disposable environment before using any procedure on valuable systems.

The author and publisher make no guarantee that a particular environment, build, service, or deployment will operate without fault. This book does not substitute for security assessment, database recovery planning, legal advice, or vendor support.

Trademarks and affiliations

Docker and Docker Hub are trademarks or registered trademarks of Docker, Inc. Other product names belong to their respective owners. Identification of these products does not imply sponsorship or endorsement.

Table of Contents

Preface
PART I
GET DOCKER RUNNING AND TAKE CONTROL
Chapter 1 — Install Docker and Run Your First Container
Chapter 2 — Understand Docker Images, Containers, and the Engine
Chapter 3 — Work Confidently with the Docker Command Line
Chapter 4 — Control Container Lifecycles and Diagnose First Failures
PART II
CREATE SMALL, REPRODUCIBLE DOCKER IMAGES
Chapter 5 — Write Your First Dockerfile and Package a Website
Chapter 6 — Containerize a Python API the Right Way
Chapter 7 — Build Faster and Smaller with BuildKit
Chapter 8 — Make Images Portable, Traceable, and Reproducible
PART III
CONNECT SERVICES AND KEEP THEIR DATA
Chapter 9 — Connect Containers with Docker Networking
Chapter 10 — Keep Data Safe with Volumes and Bind Mounts
Chapter 11 — Build Your First Multi-Container Compose Stack
Chapter 12 — Create Repeatable Development Environments
PART IV
ENGINEER A COMPLETE CONTAINERIZED APPLICATION
Chapter 13 — Expand DeployBoard into a Real API with Contracts and Authentication
Chapter 14 — Add PostgreSQL, Migrations, and Persistent Records
Chapter 15 — Add Redis and an Asynchronous Worker
Chapter 16 — Deliver a Complete Local Application Stack
PART V
SECURE IMAGES, CONTAINERS, AND SECRETS
Chapter 17 — Understand Docker's Security Model and Threat Surface

Chapter 18 — Harden Containers with Least Privilege
Chapter 19 — Handle Secrets and Configuration Without Leaks
Chapter 20 — Inspect Image Risks and Build Supply Chain Evidence
PART VI
TEST, OBSERVE, AND IMPROVE RELIABILITY
Chapter 21 — Test Docker Applications Before You Ship
Chapter 22 — Observe Containers, Services, and Resource Usage
CHAPTER 23
Troubleshoot Startup, Network, and Storage Failures
Chapter 24 — Optimize Build, Runtime, and Operational Performance
PART VII
PUBLISH, AUTOMATE, AND DEPLOY RELEASES
Chapter 25 — Publish and Manage Container Images in Registries
Chapter 26 — Build a Docker CI Pipeline with GitHub Actions
Chapter 27 — Prepare a Linux Server for Reliable Docker Deployment
Chapter 28 — Deploy DeployBoard with HTTPS and Controlled CI/CD
PART VIII
OPERATE, RECOVER, AND SHIP WITH CONFIDENCE
Chapter 29 — Back Up and Restore Containerized Application Data
Chapter 30 — Update Applications and Roll Back Failed Releases
Chapter 31 — Operate Docker Hosts and Respond to Incidents
Chapter 32 — Capstone: Ship and Operate a Secure DeployBoard Release
APPENDICES
Appendix A — Docker Installation and Recovery by Platform
Appendix B — Docker Command-Line Quick Reference
Appendix C — Dockerfile, BuildKit, and Buildx Reference
APPENDIX D — Compose Configuration Reference and Patterns
Appendix E — Complete DeployBoard Source Code and Reproducibility Kit
Appendix F — Cross-Platform Terminal and File-System Guide
APPENDIX G —Troubleshooting Decision Trees

Appendix H — Security and Production Readiness Checklists
Appendix I — Backup, Restore,Deployment, and Rollback Runbooks
Appendix J — Glossary, Version Notes, and Official References

Preface

From Your First Container to a Production Release

Your first successful docker run command is satisfying. A container starts, a port opens, a page appears, and an application that once demanded a long setup procedure now fits behind a small number of repeatable instructions. But a working container is not yet a working delivery process—and a working delivery process is not yet an operable service.

A release becomes dependable only when someone can identify the image that was deployed, explain what data must survive replacement, verify the application's behavior, trace failures from evidence, protect its secrets, and recover after an update goes wrong. These are not exotic concerns reserved for large platforms. They appear the first time an application holds data another person cannot afford to lose.

This book uses a single evolving project, DeployBoard, to join those responsibilities together. You begin with the Docker CLI and a small self-contained application. You build images, connect services, add PostgreSQL and a background worker, shape environments with Compose, enforce access rules, test failures, publish releases, and operate a hardened single-host deployment. Every major choice has a consequence you can observe.

What "production-ready" means in this book

The phrase describes a disciplined target, not a certification. A production-ready candidate has an identifiable image, documented configuration, restricted exposure, protected credentials, tested application behavior, persistent data with a recovery path, health evidence, and an operator who knows when to stop or roll back. It is not a promise that a Compose deployment is appropriate at any scale or that one security checklist eliminates all risk.

The journey is deliberate. Early examples trade complexity for clarity; later chapters replace teaching shortcuts with stricter contracts and operational controls. A local HTTP endpoint on 127.0.0.1 is a learning boundary, not a model for public Internet traffic. A development database is not a backup strategy. A successful unit test is not the same as an end-to-end recovery drill.

The promise and the limits

You will not need an external code repository to reconstruct the exercises. Commands, complete public file listings, sample data, checks, and expected observations are supplied in the chapters and appendices. You will still need a functioning Docker environment, Internet access for initial image retrieval where applicable, and enough care to record the results that only your machine can produce.

At the end, the outcome is more useful than a screen full of green containers: you will be able to explain what you shipped, why you trust its current state, how to change it, and what you will do when it fails.

Who This Book Is For

This handbook begins at the command line and advances to the responsibilities of a developer-operator. It is intended for developers learning containers, students building their first full-stack service, technical founders shipping a single-server application, and junior administrators who want a practical bridge from “it runs” to “I can operate it.”

You can start here if you can install software, create files, use a terminal, and follow short code examples. Prior Docker knowledge is not assumed. Basic HTTP, Python, and SQL familiarity helps later, but unfamiliar concepts are introduced in context.

You should expect to do real work: edit files, investigate logs, validate Compose configuration, read test failures, and perform a restore drill. This is not a command-only reference; it is a guided engineering practice.

The scope is intentional: single-host Linux container deployments, with Windows and macOS supported as learning workstations. Multi-node orchestration and large-scale platform administration are directions for further study, not hidden requirements.

What You Will Build

DeployBoard grows from a small web/API exercise into a multi-service application with authenticated HTTP endpoints, durable records, asynchronous work, a local reverse proxy, tests, a registry release, a CI pipeline, and recovery runbooks. It exists to make Docker decisions visible rather than to be a finished commercial product.

Book stage	Practical output
I · Control	A working Docker installation and a repeatable command-line inspection habit
II · Images	Reproducible application images, BuildKit caching, and multi-stage builds
III · Services	Private networks, durable volumes, and a multi-container Compose stack
IV · Application	A protected API, PostgreSQL-backed records, Redis-supported work, and a proxy
V · Security	Restricted privileges, protected secrets, and image supply-chain evidence
VI · Reliability	Automated checks, useful logs, controlled failure drills, and measured tuning
VII · Release	Versioned images, registry publication, CI, and a prepared Linux host
VIII · Operation	Backups, restoration, upgrades, rollback decisions, and final acceptance evidence

SUCCESS CONDITION You can rebuild the project, prove its most important behaviors, show why a release is safe enough for its intended environment, and recover without guessing.

How to Use This Book

Read the book in sequence for the first pass: the same application is revised as the architecture grows. A later Compose file can supersede an earlier classroom version, and security hardening may change the way commands are run. When a chapter says to replace a file, use that chapter’s complete listing rather than merging snippets by intuition.

A reliable chapter routine

- Read the outcome and prerequisites before typing commands.
- Create or replace files exactly as shown; keep the project directory and environment named in the chapter.
- Run the build and verification steps before changing the exercise.
- Use the controlled break/fix section only in a disposable lab, not against real data.
- Save the evidence: command, expected result, observed result, and repair.
- Complete the checkpoint before moving to the next dependency.

When a command behaves differently on your system, record the actual Engine, Compose, Buildx, operating system, architecture, and image digest first. Then follow the diagnostic sequence instead of restarting everything at random.

The BUILD → VERIFY → BREAK → REPAIR → SHIP Learning Loop

Each stage answers a different question. The cycle turns each chapter into a small release rehearsal, so understanding is shown through evidence rather than through the appearance of a running container.



FIGURE FM.1 A repeatable method, not an invitation to damage valuable systems.

Stage	A question you must answer
BUILD	What did I change, and what exact files or images contain that change?
VERIFY	Which command or test proves the intended behavior?
BREAK	Which safe failure exposes an assumption or dependency?
REPAIR	What evidence identified the cause, and what changed in the fix?
SHIP	Can another operator identify, launch, test, and recover this release?

A failed test is useful when its cause is isolated and documented. A pass without a traceable build or an observable outcome is weaker evidence than it looks.

Meet DeployBoard: The Project That Grows with You

DeployBoard is a compact operational tracking service built to show real container boundaries. Users create and update records through a Python API; PostgreSQL keeps durable state; a worker handles background jobs; Redis supports limited internal coordination; and Nginx provides a consistent local entry point. Docker images package the processes, while Compose names the services and their networks, mounts, health checks, and configuration.

Service	Role	Data or boundary

Service	Role	Data or boundary
api	FastAPI HTTP routes; validation and authentication	Depends on a reachable database
db	PostgreSQL transactional data	Named volume; not a published port
worker	Handles deferred jobs	Must be safe to restart and reprocess
redis	Optional short-lived coordination signals	Internal network; never a system of record
migrate	Applies controlled schema changes	Runs as a bounded maintenance step
proxy	Nginx local ingress	Bound to loopback for the lab

How the project changes by chapter

Do not assume every intermediate chapter reproduces the final Appendix E source tree byte for byte. Early API work may use a small test-oriented storage adapter; later chapters introduce PostgreSQL migrations, consolidated Compose services, hardening, and release scripts. Each chapter explains the files it changes. Appendix E supplies the canonical complete capstone inventory.

What the service deliberately does not claim

The reference system is a teachable single-host release, not a high-availability platform, public identity provider, fully managed secret store, or Kubernetes installation. A successful local Nginx connection is not an HTTPS configuration. Real external publication requires a trusted TLS path, exposed-port audit, rate-limiting decision, host security review, and recorded acceptance tests.

Software, Hardware, and Account Requirements

Your workstation should be capable of running Linux containers reliably and have enough free space for images, database volumes, build cache, and backups. The table separates official platform checks from comfortable lab planning estimates.

Item	Working recommendation
Processor	Supported 64-bit CPU with hardware virtualization enabled where required; x86-64 or ARM64 subject to image support
Memory	8 GB is a practical minimum for many supported Docker Desktop configurations; 16 GB is more comfortable for the full stack
Disk space	Plan at least 20 GB of free working space; retain separate capacity for backups and images
Software	Docker Engine + CLI + Compose v2 plugin; Buildx; terminal; text editor; browser; Git for CI chapters
Learning languages	Python 3.12 reference; SQL and Bash introduced where needed
Accounts	None required for most local labs after prerequisites are downloaded; registry and GitHub accounts for publication/CI
Cloud	Optional Linux host, domain, DNS management, SSH access, and a payment method if using a paid provider

PRE-FLIGHT Confirm host architecture, free disk space, virtualization, current Docker support requirements, and available ports before starting the first lab. Do not disable endpoint protection or host firewall simply to make a container start.

Supported Docker Versions and Platform Differences

This book uses modern Docker Engine, the Compose v2 command form (docker compose), and Buildx/BuildKit. The complete DeployBoard reference sets Compose v2.20+ as its pedagogical baseline; it is not a promise that every command, platform, or feature works unchanged with the oldest qualifying build.

```
docker version
docker compose version
docker buildx version
docker info
```

Workstation path	Important difference to verify
Windows + Docker Desktop / WSL 2	WSL 2, current supported Windows release, enabled virtualization, path mapping, PowerShell vs Bash, and Linux container mode
macOS + Docker Desktop	Apple silicon vs Intel architecture, Linux VM resource limits, file sharing, and multi-platform image availability
Linux + Docker Engine	Compose/Buildx plugins, daemon and group privileges, rootless limitations, file UID/GID ownership, and host firewall rules

Use the official Docker installation pages to confirm current requirements before relying on an older printed number. Docker Desktop licensing may vary by business size and use case. Appendix A covers platform recovery, Appendix F covers shell and filesystem differences, and Appendix J records reference versions and authoritative documentation.

Where to get help and what to verify first

Appendix B provides CLI commands; C covers Dockerfile, BuildKit, and Buildx; D covers Compose; F explains platform differences; G provides failure decision trees; H provides security and readiness checks; I provides backup, recovery, and rollback runbooks; and J identifies version and reference boundaries.

Free Local Deployment and Optional Cloud Deployment Paths

The default path is local, private, and low-cost: build images on your workstation, run the Compose stack, publish the proxy to 127.0.0.1, and test through a browser or curl. This proves application behavior without exposing a development service to the public Internet. Image downloads and Internet connectivity may be required even when the subsequent lab runs locally.

Path	Use it for	Boundary
Local, loopback-only	Chapters 1–26, security drills, most tests, and restore rehearsals	No public reachability; protect local data and host privileges
Optional Linux server	Release engineering, real network ingress, host operation, and public HTTPS	Provider costs, firewall, DNS, TLS, monitoring, and backups are your responsibility

If you do not want to pay for hosting, complete the local route and perform the release-readiness exercise without publishing a public endpoint. The engineering lessons remain valid. If you choose cloud hosting, estimate usage and set provider budget alerts before creating resources.

Cost, Security, and Lab Safety Notes

- Use synthetic records and generated practice credentials only. Never paste real API tokens or production passwords into commands, screenshots, issue posts, or source control.
- Prefer 127.0.0.1 host bindings while learning. A published 0.0.0.0 port can be reachable beyond your machine depending on network and firewall behavior.
- Treat Docker daemon access, a mounted Docker socket, privileged containers, and unrestricted bind mounts as host-security decisions—not harmless convenience settings.
- Never run docker compose down --volumes, docker volume prune, or destructive database commands against data you intend to keep. Read the target project name and resource list first.
- Run intentional fault drills under a separate Compose project name and disposable resources. Practice restoration into an isolated database; never assume a backup works because the file exists.
- Back up important data before upgrades. Record image digests, database schema assumptions, and a rollback decision before changing the deployment.
- Set cloud spending alerts and remove unused resources. Stopping a VM does not necessarily remove storage, IP-address, snapshot, or domain charges.

STOP RULE If you cannot explain what a command will modify or delete, inspect its flags and the exact target before running it. A lab is successful only when you can verify the resulting state.

Conventions Used Throughout the Book

Examples are written for readability and reproducibility. Shell commands, file paths, environment variables, filenames, images, host names, and responses are visually distinguished because each carries a different kind of instruction.

Printed form	Meaning / reader action
docker compose ps	Command to type in the named terminal
compose.yaml	Create or modify a file with this exact relative name
APP_TOKEN_READ_FILE	Environment variable name; not a secret value
<your-host>	Replace with your actual value; do not type the angle brackets
Expected: HTTP 200	Example outcome to verify, not a transcript guaranteed on every machine
STOP / CAUTION	Boundary check: understand the risk before continuing

Prompt symbols such as $ or PS> appear only when the surrounding prose needs to show a terminal context; they are not part of a command unless a listing explicitly says otherwise. Use standard straight quotes for copied commands. For Windows PowerShell versus Bash differences, consult Appendix F rather than silently changing path separators or line continuations.

Commands, Complete File Listings, and Expected-Output Conventions

Complete listings beat missing glue

When the book introduces or replaces a project file, it supplies the complete public contents needed for that checkpoint. Create directories exactly as shown, preserve indentation, and save files in UTF-8. Code snippets marked as excerpts explain a behavior and are not substitutes for

the corresponding complete file. The capstone file inventory and runnable reference are collected in Appendix E.

Three outputs to distinguish

Expected: The result a correct run should show in the stated scenario. A version string, hash, container ID, timestamp, or host path will normally differ.

Observed: What your own terminal, browser, or test actually returned. Record it before making changes if it indicates a failure.

Acceptance: The measurable condition that must hold before you proceed—for example, authorized requests succeed while unauthorized writes are rejected; records survive a recreate; or a restore drill proves recoverable data.

A minimal evidence receipt

```
Date / time (UTC): <timestamp>
Host / OS / architecture: <details>
Docker / Compose / Buildx: <versions>
Project / image digest: <identifiers>
Command or test: <exact action>
Expected / observed: <results>
Decision: PASS | FAIL | BLOCKED
Next action / owner: <follow-up>
```

A PASS requires actual evidence. If a command was skipped because Docker was unavailable, write BLOCKED or NOT RUN; do not translate an offline unit-test pass into a live deployment approval. Wherever a destructive action is taught, the chapter explains what data is at risk and how to use disposable replacements.

Start with a visible result. Finish with a recoverable one.

PART I

GET DOCKER RUNNING AND TAKE CONTROL

Chapter 1 — Install Docker and Run Your First Container

Your first deliverable is not an application. It is a workstation you can trust to run one.

A teammate gives you a website and says, “It works on my machine.” Before you can package, ship, or debug that application, you need a working container engine and a clear understanding of where the application runs. In this chapter, you install the right Docker tools, prove the engine responds, and serve a real web page from a Linux container. Nothing here requires a cloud account or a paid server.

> **YOUR CHAPTER DELIVERABLE** A verified Docker workspace and a local Nginx site reachable at http://127.0.0.1:8080. You will also know exactly how to stop it and remove only the resources you created.

Your workstation	Your route	Terminal used
Windows 10/11, supported edition	Docker Desktop + WSL 2, Linux containers	PowerShell
macOS on Apple silicon or Intel	Docker Desktop for Mac	Terminal
Ubuntu Linux, supported release	Docker Engine + Compose/Buildx plugins	Bash

Follow **one operating-system installation route** in Sections 1.4–1.6. Everyone rejoins the same container exercises in Section 1.7. If you are using a managed computer, confirm that installing software and enabling virtualization are permitted before proceeding.

1.1 Choose Your Docker Learning Path

Docker separates the application package from much of the configuration of the computer that runs it. The installation path still depends on your host operating system. Choose your route now; do not combine commands from different systems.

Decide in 30 seconds

1. **Windows:** use Docker Desktop with the WSL 2 backend. Run Windows-specific setup commands in PowerShell. Use Linux containers, which the projects in this book expect.
2. **macOS:** install Docker Desktop for the architecture of your Mac. Run shell commands in Terminal. Docker Desktop supplies the Linux environment behind the scenes.
3. **Ubuntu Linux:** install Docker Engine and its current Compose and Buildx plugins from Docker’s official apt repository. Use Bash; Linux-specific commands require `sudo` unless you deliberately configure another access model.

On Windows and macOS, Docker Desktop is a developer-facing package that includes the CLI, Engine environment, Compose, and build tools. On Linux, Docker Engine can run natively without Docker Desktop. The course uses the modern `docker compose` subcommand, not the older standalone `docker-compose` spelling.

> **BOUNDARY OF THIS CHAPTER** All examples use Linux containers. Windows containers, GPU workloads, enterprise desktops, and cluster orchestration require different setup choices and are not prerequisites.

1.2 Understand What a Container Solves

A container runs an application as a process with an isolated view of parts of the system: filesystem, networking, process IDs, and resource controls. The image supplies a repeatable starting filesystem and startup configuration. A container is a running instance of that image, not a miniature computer that automatically provides its own full operating-system kernel.

The practical benefit is consistency: instead of asking every teammate to install the same web server and libraries by hand, you can run a specified image. You still need to handle data, secrets, networking, security, and updates explicitly. Containers do not guarantee that software is bug-free or that a deployment is production-ready.

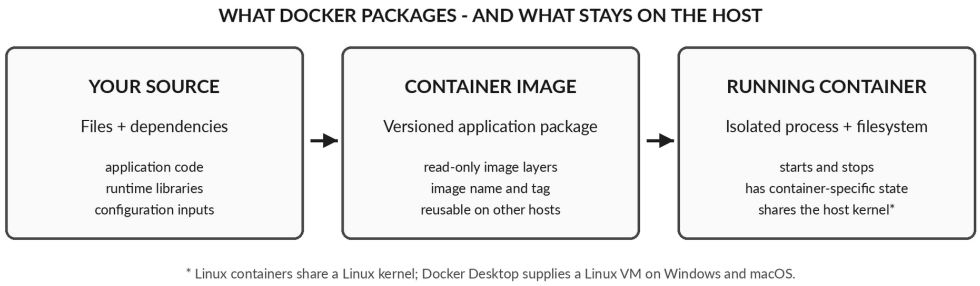


Figure 1.1 — Source becomes an image; an image can be started as a container on a compatible Docker host.

Linux containers share a Linux kernel. On a native Linux installation, that kernel is the host's. On Windows and macOS, Docker Desktop uses a lightweight Linux virtualized environment to supply it. This difference matters later when paths, permissions, and network behavior appear to differ between operating systems.

The mental model you need today

Image: the reusable package. **Container:** a process created from that package with its own writable layer and runtime settings. **Engine:** the component that creates and supervises containers. **CLI:** the tool you type commands into. **Registry:** a remote place from which Docker can obtain images. Chapter 2 will inspect these pieces in depth; today you need only enough to operate them safely.

1.3 Check Hardware, Operating System, and Permissions

Before installing anything, check the environment. Requirements evolve, so the current Docker installation pages are authoritative if a system differs from the examples below. For a comfortable learning environment, allow roughly 15–20 GB of free space beyond the operating system and leave enough memory for a browser, editor, and containers. Those are practical recommendations, not official minimums.

Platform	Critical checks before installation
Windows	Supported 64-bit Windows release; WSL 2 enabled; virtualization enabled; at least 8 GB RAM for the documented x86-64 WSL 2 route.
macOS	A currently supported macOS release; compatible Apple-silicon or Intel installer; at least 4 GB RAM (more is preferable).
Ubuntu Linux	Supported 64-bit Ubuntu release; administrator access for apt; network access to download.docker.com and image registries.

Run only the checks for your platform

Windows PowerShell — WSL checks (output varies)

```
wsl --version
wsl --status
```

If `wsl --version` does not display version information, update WSL before installing Docker Desktop. Check the Windows version with `winver`, and look for enabled virtualization under Task Manager → Performance → CPU.

macOS Terminal — CPU family and macOS version

```
uname -m
sw_vers -productVersion
```

On a Mac, `arm64` identifies Apple silicon and `x86_64` identifies Intel. Download the matching Docker Desktop build.

Ubuntu Bash — distribution, architecture, and free disk space

```
cat /etc/os-release
uname -m
df -h .
```

On Linux, confirm that the current user may run approved `sudo` commands. On any platform, a corporate VPN, restrictive proxy, or blocked registry may prevent downloads; this is a network issue rather than evidence that the Docker CLI is broken.

> **LICENSING CHECKPOINT** Docker Desktop is free for personal use, education, qualifying small businesses, and non-commercial open-source projects under its current terms. Larger enterprises and certain other uses require paid licensing. Check the current agreement before organizational deployment; Docker Engine has different licensing terms.

1.4 Install Docker Desktop on Windows with WSL 2

Use this route if you chose Windows. The commands in this section are PowerShell commands; do not paste them into a Linux shell. Docker Desktop currently recommends a per-user installation with WSL 2 for typical Linux-container development. Make sure your Windows edition and WSL version meet the current published requirements.

Prepare WSL 2

1. Open PowerShell. Run `wsl --version`. If WSL is absent, launch PowerShell as administrator and run `wsl --install`. Follow Windows prompts and restart if requested.
2. If WSL is installed but outdated, run `wsl --update`. Check the result again with `wsl --version`.
3. Confirm that hardware virtualization is enabled in firmware (BIOS/UEFI). If WSL reports that virtualization support is unavailable, resolve that system prerequisite before attempting Docker Desktop.

PowerShell — after WSL installation or update

```
wsl --version
wsl --status
```

The version numbers will differ. The important signs are a working WSL installation and WSL 2 support. If you already use a Linux distribution under WSL, you may check it with `wsl -l -v`, but Docker Desktop does not require you to install a separate Docker Engine inside that distribution.

Install and launch Docker Desktop

1. Download Docker Desktop for Windows from https://docs.docker.com/desktop/setup/install/windows-install/ , choosing the supported package for your machine. Avoid unofficial download sites.
2. Run **Docker Desktop Installer.exe**. The per-user option is the recommended default for most learners; organization policies may require the all-users mode.
3. Use the **WSL 2** backend when prompted. Complete the wizard and start Docker Desktop from the Start menu.
4. Review the license agreement and accept it only if the terms apply to your use. Allow Docker Desktop to finish starting.
5. Open Settings if necessary and confirm that Docker Desktop is configured for **Linux containers** and a supported Linux-container backend. Leave advanced resource settings at the defaults unless you have a specific problem.

Open a **new PowerShell window**, then continue in Section 1.7. A message such as "Cannot connect to the Docker daemon" usually means Docker Desktop is not started yet or the selected context points somewhere else. Do not fix that error by installing a second Docker daemon inside WSL.

> **WINDOWS EDITION NOTE** Docker Desktop installation mode, backend availability, and Windows container features depend on the Windows edition. This book uses Linux containers only. Use the current Docker compatibility table rather than assuming all Windows editions have identical feature support.

1.5 Install Docker Desktop on macOS

Docker Desktop supplies the Linux VM, daemon, CLI, Compose, and build functionality that this book needs. The installer must match your Mac's processor type, and your macOS version must still be in Docker's supported window.

1. In Terminal, run `uname -m` . Choose the **Apple silicon** installer if it prints `arm64` , or the **Intel** installer if it prints `x86_64` .
2. Download the corresponding `.dmg` from https://docs.docker.com/desktop/setup/install/mac-install/ .
3. Open `Docker.dmg` and drag **Docker.app** into the Applications folder. Launch Docker from Applications.
4. Approve macOS permissions only after reading the prompts. Review the subscription agreement, then allow the Docker Engine environment to finish starting.
5. Open Terminal again and continue with the verification commands in Section 1.7.

macOS Terminal — processor type and selected Docker context

```
uname -m
docker context show
```

Some Intel-targeted tooling may need translation support on Apple silicon; Rosetta 2 is not a blanket prerequisite for running Linux ARM64 containers. You will learn about image architectures and multi-platform builds in Chapter 8.

> **MACOS REALITY CHECK** A Linux container is not running against the macOS kernel. Docker Desktop provides the Linux environment. Avoid treating paths inside a container as though they were ordinary macOS folders; bind mounts and file sharing are covered in Chapter 10.

1.6 Install Docker Engine and the Compose Plugin on Linux

This route uses Ubuntu. It installs Docker Engine, the CLI, containerd, Buildx, and Compose from Docker's official apt repository. If your machine already runs important containers or was provisioned by an employer, stop and review its existing package sources and services before changing packages.

Check for conflicts, then add the official repository

Ubuntu may already have distro-packaged `docker.io`, `containerd`, or other components. Docker warns that such packages can conflict with its official packages. On a previously used system, consult the official conflict-removal list and back up anything important; do not remove existing software blindly. On a fresh supported Ubuntu host, use these commands:

Ubuntu Bash — install prerequisites and repository signing key

```
sudo apt update
sudo apt install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
  -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

Ubuntu Bash — configure Docker's stable apt source

```
sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
sudo apt update
```

The lines between `<<EOF` and `EOF` form one shell command. Paste the complete block into Bash, not PowerShell. If apt cannot find the repository for your Ubuntu release, check its official support status and the printed codename instead of replacing it with a guessed value.

Install the supported Docker packages

Ubuntu Bash — install and verify the native engine

```
sudo apt install docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin
sudo systemctl is-active docker
sudo docker run --rm hello-world
```

`systemctl is-active` should print `active`. The test container should print a message beginning with **Hello from Docker!** and then exit. The first run needs network access to download the image. If the service is not running, check `sudo systemctl status docker` and, where appropriate, start it with `sudo systemctl start docker`.

Choose a safe CLI permission model

By default, ordinary Linux users may need `sudo docker ...`. For this chapter you can continue using `sudo` before Docker commands. Some developers add their account to the `docker` group, but Docker explicitly warns that group membership confers root-level privileges; it is **not** an unprivileged workaround.

Optional on a personal machine — grant highly privileged Docker group access

```
sudo usermod -aG docker "$USER"
# Then sign out completely and sign back in.
id -nG
```

Do not use the optional group command on a shared or managed host without authorization. Rootless Docker is another approach, with its own limitations; we evaluate it in the security section of the book. If you stay with `sudo`, prepend `sudo` to the shared `docker` commands in Sections 1.7–1.12.

1.7 Verify the Engine, CLI, Compose, and Buildx

Now everyone uses the same Docker CLI vocabulary. Windows and macOS learners should run the commands in PowerShell or Terminal, respectively. Ubuntu users without configured Docker access should prefix commands with `sudo`. Copy the following commands one line at a time so you can identify which check fails.

All platforms — CLI, daemon, plugin, and target checks

```
docker version
docker info
docker compose version
docker buildx version
docker context show
```

Command	What proves success
docker version	Both Client and Server sections appear; versions need not be identical.
docker info	Engine information is shown instead of a daemon connection error.
docker compose version	A Docker Compose v2 version is displayed.
docker buildx version	Buildx reports its installed plugin version.
docker context show	A context name is shown; it targets the engine you intend to use.

The selected context often differs: Docker Desktop may use `desktop-linux`, while native Linux commonly uses `default`. Neither name by itself proves a fault. If the server is unreachable, verify Docker Desktop is running or the Linux `docker` service is active.

All platforms — Linux-container target and end-to-end test

```
docker info --format "{{.OSType}}"
docker run --rm hello-world
```

The operating-system type should be `linux`. The one-off test should include **Hello from Docker!** followed by explanatory text. `--rm` removes the test container after it exits; it does not automatically delete the downloaded image. If a corporate proxy blocks the pull, the problem may be registry access, not a failed installation.

> **THE PROOF THAT MATTERS** A successful `docker --version` alone is insufficient: it proves the client exists, not that it can contact an Engine. The decisive test is a successful daemon response followed by a successfully run container.

1.8 Run, Open, and Stop Your First Web Container

You are going to run an actual web server using the official Nginx Alpine image. First, start it **without publishing a host port**. This deliberate choice lets you observe the container before exposing any route to it.

All platforms — start and inspect the web server

```
docker run -d --name first-web nginx:alpine
docker ps --filter name=first-web
docker logs first-web
```

`-d` starts the container in detached mode, `--name` gives it a predictable identifier, and `nginx:alpine` selects the image. The first command returns a long container ID. `docker ps` should show `first-web` with a running status. The `PORTS` column may show `80/tcp`, but that is **not** a published host port.

Open a shell inside the running container

Interactive container shell — execute lines after the shell opens

```
docker exec -it first-web sh
wget -qO- http://127.0.0.1/ | head -n 5
exit
```

Run the first line in your host terminal. The next two lines execute **inside the container**, where the Alpine image provides a small shell and BusyBox tools. The output should begin with HTML from the default Nginx page. `exit` returns you to your host terminal; it does not stop the web server.

All platforms — stop, then compare running and all-container listings

```
docker stop first-web
docker ps --filter name=first-web
docker ps -a --filter name=first-web
```

After stopping, `docker ps` no longer lists a running `first-web`, but `docker ps -a` still shows it as exited. The container was stopped, not removed. This distinction is fundamental when you later handle data and deployments.

1.9 Publish a Container Port and Test It in a Browser

A container can listen on its own port without your browser being able to reach it. Docker must publish a route from a host address and port to the container's listening port. We will bind only to `127.0.0.1` so this exercise is accessible from the local computer, not intentionally from every network interface.

Docker does not let you add a new published-port setting to an existing container using `docker start`. Remove the stopped, disposable `first-web` container from Section 1.8 and create a replacement with the correct port mapping.

All platforms — recreate with local-only published port

```
docker rm first-web
docker run -d --name first-web -p 127.0.0.1:8080:80 nginx:alpine
docker port first-web
```

Open **http://127.0.0.1:8080** in your browser. You should see the Nginx welcome page. A browser page is the application-level proof that the container is running, its internal server is listening, and Docker's host-port mapping works.

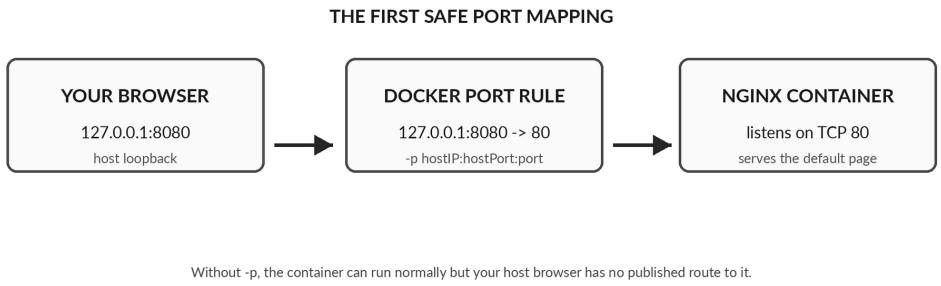


Figure 1.2 — A localhost-only port binding connects your browser to TCP port 80 inside Nginx.

Test without relying on the browser

macOS or Linux Terminal — inspect the HTTP response headers

```
curl -I http://127.0.0.1:8080/
```

Windows PowerShell — request the local webpage

```
Invoke-WebRequest http://127.0.0.1:8080/
```

A successful response normally has HTTP status `200`. If port `8080` is already occupied, use `8081` on the host side (`127.0.0.1:8081:80`) and update the address everywhere in this exercise. Do not change the container-side `80` for this image.

Stop the container and refresh the browser: the page should become unavailable. Start it again and confirm that the same address works. These actions demonstrate that the port mapping belongs to the container configuration and survives an ordinary stop/start cycle.

All platforms — prove stop and restart behavior

```
docker stop first-web
docker start first-web
docker ps --filter name=first-web
```

> **SECURITY FACT TO REMEMBER** Using `-p 8080:80` without a host IP commonly publishes on all host interfaces. Docker's Linux firewall behavior can also differ from expectations based on UFW. We deliberately use `127.0.0.1` here; production exposure receives its own verified treatment in Chapter 27.

1.10 Understand What Happened Behind the Command

Look back at one line: `docker run -d --name first-web -p 127.0.0.1:8080:80 nginx:alpine` . It hides a useful sequence of actions. Understanding that sequence makes later failures easier to diagnose.

1. **Resolve the image:** Docker checks whether the requested image is available locally. If not, it tries to pull the image from its configured registry.
2. **Create the container:** the Engine combines read-only image content with container-specific configuration and a writable layer.
3. **Start the process:** Nginx runs as the container's main server process. The container normally stays alive while that process runs.
4. **Apply network settings:** Docker arranges a port mapping from the selected host loopback address and port to TCP port 80 inside the container.
5. **Return control:** detached mode leaves Nginx running while your host terminal becomes available for other commands.

You can inspect pieces of that sequence without guessing. The following commands display the image, port mapping, status, and recent logs. The full inspection toolbox appears in Chapter 3.

All platforms — connect observable evidence to the running system

```
docker image ls nginx
docker inspect --format "{{.State.Status}}" first-web
docker port first-web
docker logs --tail 10 first-web
```

Your `docker image ls` result can differ in tag or size depending on the image version and architecture. Use the container name to inspect **that container**, not an unrelated service already running on your computer.

1.11 Clean Up Without Removing Important Data

Good Docker hygiene is targeted. You should be able to identify the exact resource you created, stop it, remove its container, and leave unrelated applications untouched. You do **not** need to reset Docker Desktop or run a broad pruning command to finish this exercise.

All platforms — remove only the demonstration container

```
docker stop first-web
docker rm first-web
docker ps -a --filter name=first-web
docker image ls nginx:alpine
```

The container-filtered listing should no longer show `first-web`. The downloaded image may remain cached, which makes your next run faster. An image is not the same as a container. You may remove this image later with `docker image rm nginx:alpine` **only if** you have verified no other container depends on it and you actually want to reclaim the space.

Command	Typical effect	Use in this chapter?
docker stop NAME	Stops the named running container.	Yes
docker rm NAME	Removes a stopped named container.	Yes
docker image rm IMAGE	Deletes an image when it is no longer in use.	Optional, after checking
docker system prune -a --volumes	Potentially removes many unused objects and volumes.	No — avoid

> **DATA SAFETY** Nothing in this chapter stores application records in a named volume. Later chapters will. Never run cleanup commands with `--volumes` on an unfamiliar workstation; volume removal can destroy data you meant to keep.

1.12 Checkpoint: Prove Your Docker Workspace Works

The last task is a small acceptance test. Imagine a colleague asks whether your machine is ready for the next project. Instead of saying “Docker seems installed,” collect the following evidence.

A reproducible acceptance test

All platforms — execute on the Docker engine selected in Section 1.7

```
docker compose version
docker buildx version
docker run --rm hello-world
docker run -d --name chapter01-proof -p 127.0.0.1:8080:80 nginx:alpine
```

```
docker ps --filter name=chapter01-proof
docker port chapter01-proof
```

Visit **http://127.0.0.1:8080** and confirm the Nginx page appears. If you used host port `8081` earlier, substitute that port here too. On native Linux, prepend `sudo` unless your account has deliberately configured Docker Engine access.

Proof	Pass condition
Engine access	docker run --rm hello-world completes successfully.
Required tools	Compose and Buildx both report versions.
Runtime process	chapter01-proof appears with a running status.
Network route	docker port lists the localhost mapping you chose.
Application check	The browser renders the Nginx welcome page.

When a check fails

Symptom	First investigation
docker: command not found	Reopen the terminal; verify installation and PATH; on Linux check the package install.
Cannot connect to Docker daemon	Start Docker Desktop or inspect `sudo systemctl status docker`; check the active context.
Permission denied to Docker socket (Linux)	Use authorized `sudo docker ...` or review your explicit access model.
Image pull times out	Check internet, proxy, VPN, DNS, registry availability, and authentication limits.
Port 8080 already allocated	Choose host port 8081, then adjust the URL; do not kill unrelated processes.
Browser cannot load the page	Verify `docker ps`, `docker port`, logs, and the exact loopback URL.

After your browser test, remove the checkpoint container:

All platforms — finish with a clean workspace

```
docker stop chapter01-proof
docker rm chapter01-proof
docker ps -a --filter name=chapter01-proof
```

At this point, you have a working Engine, a verified CLI, Buildx and Compose, a successful pull-and-run operation, a web server reached through a deliberate port mapping, and a safe cleanup procedure. **Keep the installed tools and cached image; Chapter 2 begins with an image and a container you can inspect.**

Chapter 2 — Understand Docker Images, Containers, and the Engine

You already know how to run a container. Now learn what Docker actually created, which component did the work, and what you can verify before trusting the result.

In Chapter 1, a web server appeared in your browser. That proved the basic installation works. It did not yet tell you where the server files came from, why a container has an identity separate from its image, or what the Docker daemon is allowed to do. Those distinctions are the foundation of every project that follows.

The workbench for this chapter

We will use a small, official Alpine Linux image and a deliberately named container. No application source files, paid services, account login, or Compose configuration are needed. You can run the commands in PowerShell on Windows, Terminal on macOS, or Bash on Linux. Use the Docker context you verified in Chapter 1.

```
docker version
docker context show
docker pull
alpine:3.22
```

> **EXPECTED RESULT** The client and server portions of docker version are visible, docker context show names your active context, and the image pull completes. Version numbers and the reported digest vary over time and across architectures.
>
> **SAFETY BOUNDARY** Every container created here is named ch2-... . The cleanup instructions remove only those named lab containers. Do not use broad prune commands on a machine that holds important workloads.

2.1 Distinguish Containers from Virtual Machines

A virtual machine emulates a computer-like environment and boots a guest operating system with its own kernel. A Linux container is a set of

processes isolated by Linux kernel features while using a Linux kernel supplied by its execution environment. The container may have its own filesystem, process view, and network namespace, but it is not a separate kernel.

This distinction explains why a container usually starts faster and uses less overhead than a general-purpose VM, but it does not make containers inherently secure or interchangeable with VMs. The right choice depends on the isolation required, the operating systems involved, and the operational constraints.

Question	Linux container	Virtual machine
What runs?	Isolated application processes	Guest operating system plus processes
Kernel?	Shares the environment's Linux kernel	Boots its own guest kernel
Boot path?	Starts configured processes	Typically boots a complete OS
Typical use?	Package and operate services consistently	Different kernels, strong workload boundaries, full OS
Security?	Depends on kernel and runtime configuration	Depends on hypervisor and VM configuration

> **IMPORTANT EXCEPTION** On Windows and macOS, Docker Desktop supplies a Linux execution environment using WSL 2 or a Linux VM. You are still running Linux containers, even though the physical computer does not use Linux as its everyday desktop OS.

For DeployBoard, containers let us package an API and its dependencies consistently. In later chapters, we will use multiple containers to keep the API, database, and worker independent. We are not attempting to build a virtual machine for each service.

2.2 Understand the Docker Client, Daemon, and API

When you enter docker ps, your terminal is not scanning processes directly. The Docker command-line client sends a request to an Engine API endpoint. The daemon, usually dockerd, performs the requested work using lower-level runtime and image-management components. The client then displays the response.

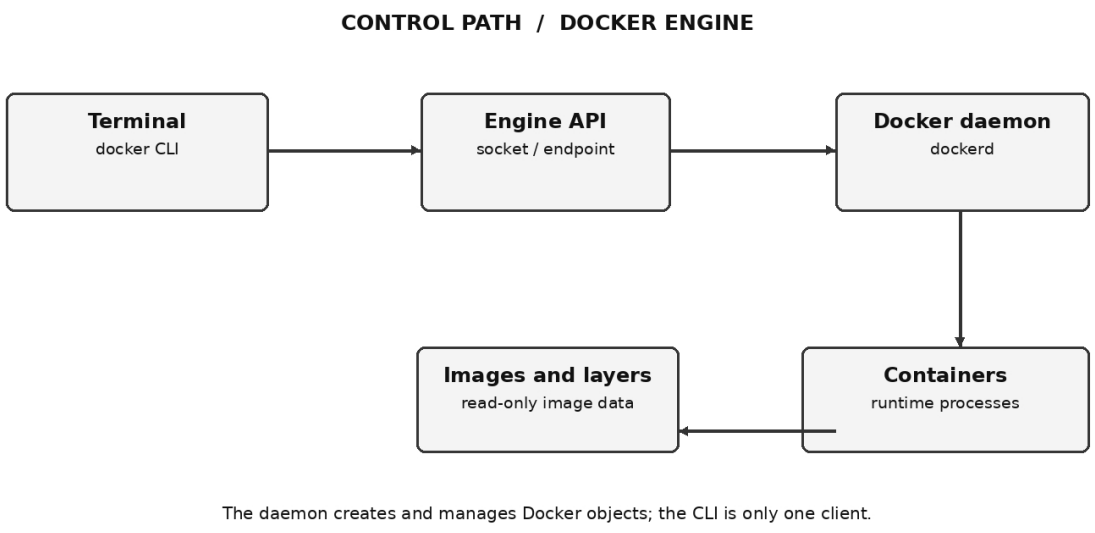


Figure 2.1 — The Docker CLI sends requests through the Engine API; the daemon manages images, containers, networks, and volumes.

The distinction is useful when diagnosing failures. A successful docker --version only proves that a client executable exists. A functioning Docker Engine also requires a reachable, authorized daemon. If docker version shows a client but reports a server connection error, inspect the active context and daemon service before reinstalling anything.

```
docker --version
docker version
docker context ls
docker info --format
'{{.OSType}}/{{.Architecture}}'
```

EXPECTED RESULT docker --version prints the CLI version. docker version reports client and server information if connected. docker context ls marks the active context with an asterisk. For these Linux-container labs, docker info normally reports linux and the daemon architecture.

The API may be exposed over a local Unix socket, a Windows-specific named pipe, or a configured network endpoint. A remote Docker context can make a command affect a different machine. Always check the current context before creating, removing, or publishing resources.

SECURITY IMPLICATION Access to a conventional rootful Docker daemon is highly privileged. Do not expose an unauthenticated Docker API over TCP or mount the Docker socket into untrusted containers. Chapter 17 treats this as a security boundary, not merely a setup detail.

2.3 Trace the Image-to-Container Lifecycle

An image is a reusable package; a container is one runtime instance created from it. One image can produce many containers. Each container receives its own runtime configuration, identifier, and writable layer. The image does not become a container by changing its identity.

Lab: separate creation from execution

Run the next commands one at a time. The sleep process gives you enough time to inspect the state before the container exits.

```
docker container create --name ch2-workspace alpine:3.22 sleep
3600
docker container ls -a --filter name=ch2-workspace
docker container inspect --format '{{.State.Status}}' ch2-workspace
```

EXPECTED RESULT The first command prints a container ID. The listing shows ch2-workspace with status Created, and the formatted inspection prints created. No application process is running yet.

```
docker container start ch2-workspace
docker container inspect --format '{{.State.Status}}' ch2-
workspace
docker container stop ch2-workspace
docker container inspect --format '{{.State.Status}}' ch2-
workspace
```

EXPECTED RESULT The formatted status changes from running to exited. Stopping the container does not delete it, and it does not delete the Alpine image.

A docker run command combines the usual create-and-start sequence. In later projects, Compose will manage several such service instances. Learning the underlying states now prevents confusion when a service has been created but has not started, or when a process exits while its container record remains.

2.4 Understand Registries, Repositories, Tags, and Digests

The name alpine:3.22 identifies a repository plus a tag. Because no registry host or namespace was supplied, Docker resolves it as a Docker Official Image on Docker Hub. A registry stores images; a repository groups related image references; a tag is a readable pointer that may be retargeted; and a digest identifies content.

Element	Example	Practical meaning
Registry	docker.io	Where image content is hosted

Element	Example	Practical meaning
Namespace / repository	library/alpine	Which image project or owner
Tag	3.22	Readable release-family reference; may change
Digest	sha256:...	Content-addressed reference used for pinning
Local image ID	sha256:...	Identifies the stored image configuration

```
docker image ls alpine
docker image inspect alpine:3.22 --format '{{.Id}}'
docker image inspect alpine:3.22 --format '{{json
.RepoDigests}}'
```

The image ID and the digest shown in RepoDigests are not necessarily the same hash. A pulled reference can involve a multi-platform manifest or a platform-specific manifest, while the local image ID identifies its configuration. If RepoDigests is empty for a locally built image, that does not mean the image is corrupt or uninspectable.

> **RELEASE DISCIPLINE** Tags are convenient for learning and tracking a supported release family; a digest pin is preferable when you need to redeploy exactly the same artifact. Pinning also freezes a security state, so pinned images still require deliberate update and patch processes.

You will learn to build and tag your own images in Chapters 5–8. For now, record the displayed image reference and understand what it does—and does not—promise about reproducibility.

2.5 Inspect Image Metadata and Container Configuration

The fastest way to stop guessing about a container is to inspect it. Images describe potential starting defaults. Containers hold the configuration and observed state of one instance. That distinction matters when a command overrides the image's default user, startup command, port publishing, or resource limits.

Inspect the image

```
docker image inspect alpine:3.22 --format
'{{.Os}}/{{.Architecture}}'
docker image inspect alpine:3.22 --format '{{json .Config.Cmd}}'
docker image inspect alpine:3.22 --format '{{.Config.User}}'
```

INTERPRETATION The OS and architecture describe the image variant selected for this daemon. The default command is an image setting, not proof that your specific container used it. An empty Config.User typically means no explicit user was set in the image; do not interpret that as proof the process runs without root privileges.

Inspect the actual container

```
docker container inspect --format '{{.Image}}' ch2-workspace
docker container inspect --format '{{json .Config.Cmd}}' ch2-workspace
docker container inspect --format '{{.HostConfig.Privileged}}' ch2-workspace
docker container inspect --format '{{.State.ExitCode}}' ch2-workspace
```

You should see an image identifier, the sleep command you supplied, false for Privileged, and an exit code. A stopped container is still inspectable. Compare image defaults with container overrides whenever behavior is surprising.

Reading raw JSON: docker image inspect alpine:3.22 and docker container inspect ch2-workspace return detailed JSON. Formatted fields keep small investigations readable without requiring an extra JSON-processing tool.

2.6 Explore Namespaces, Cgroups, and Filesystem Layers

Linux namespaces govern what a container process can see: process IDs, mount points, networking, hostnames, and other resources may have separate views. Control groups (cgroups) account for and can constrain CPU, memory, and other resources. Filesystem layers provide the image's starting files and a separate place for runtime changes.

Mechanism	What it changes	What it does not guarantee
PID namespace	Which processes are visible	Protection against all kernel flaws
Network namespace	Interfaces, routes, and ports seen	No external access unless configured
Mount namespace	Filesystem mount view	Safety of exposed host bind mounts
User namespace	Maps user/group identities	That every image is automatically non-root
Cgroups	Accounting and supported limits	Fairness without correct resource settings

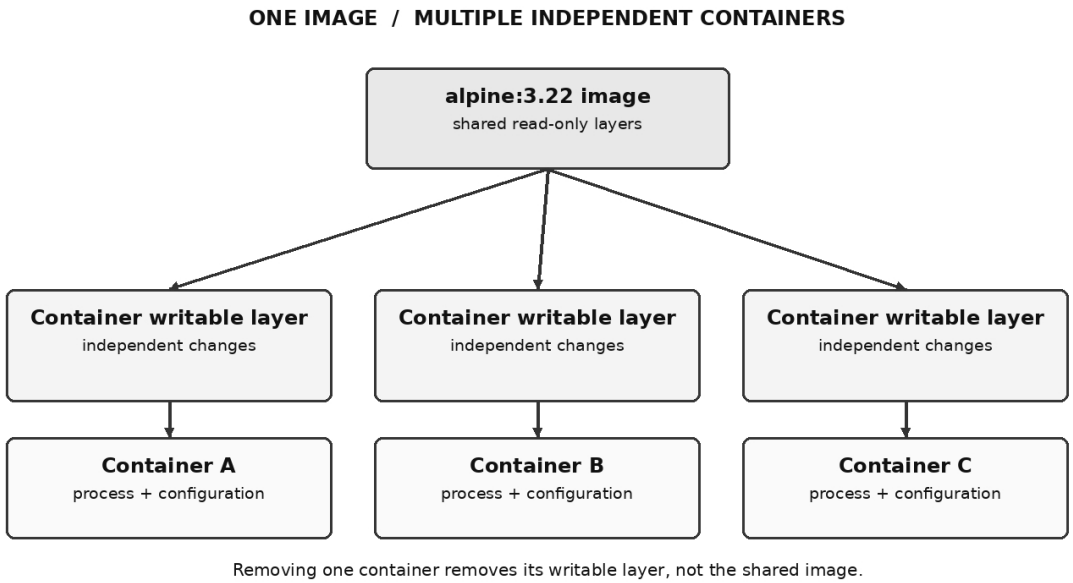


Figure 2.2 — Multiple containers can share the same image while keeping separate writable layers and process configurations.

Lab: demonstrate the boundary with a disposable file

```
docker run --rm alpine:3.22 sh -c 'echo temporary > /tmp/ch2-proof'
docker run --rm alpine:3.22 sh -c 'test ! -e /tmp/ch2-proof && echo file-
absent'
```

EXPECTED RESULT The second command prints file-absent. Each run created a new container and removed it on exit. The first file belonged only to that first container's writable layer. This is not a test of named volumes, which Chapter 10 introduces.

Storage implementations vary by Docker Engine version and configuration. Classic overlay2 documentation explains copy-on-write effectively, but new Engine installations can use a containerd image store and snapshotters. For this book, the dependable model is image content plus container-specific writes—not a hardcoded directory under /var/lib/docker.

2.7 Understand Container Processes and Exit Codes

A container's running state depends on its configured main process. When that process exits, the container stops even if the image itself is valid. Docker records a numeric exit code that you can inspect after the fact. For ordinary Unix programs, zero conventionally means success; a nonzero code means a failure or another intentional condition defined by the process.

Lab: create a predictable failure

```
docker container create --name ch2-exit alpine:3.22 sh -c 'exit
7'
docker container start ch2-exit
docker container inspect --format '{{.State.Status}}' ch2-exit
docker container inspect --format '{{.State.ExitCode}}' ch2-
exit
```

> **EXPECTED RESULT** The container status becomes exited and the exit code is 7. The code came from our explicit shell command, not from a defect in the Alpine image.

Exit status is evidence, not a full diagnosis. Exit 137 often indicates SIGKILL and can accompany an out-of-memory kill, but it is not by itself proof of memory exhaustion. A 143 code commonly reflects SIGTERM. Docker command-level errors such as 125, 126, and 127 have separate meanings. In Chapter 4, we will combine status, signals, timestamps, and logs before choosing a repair.

The first process in a container has special process-management responsibilities on Linux. Startup scripts, signal forwarding, and graceful shutdown matter once the container runs a real API or database. For this chapter, the goal is to know where to find the main process and its exit result.

```
docker container inspect --format '{{.Path}}' ch2-workspace
docker container inspect --format '{{.State.FinishedAt}}' ch2-
exit
docker container logs ch2-exit
```

> **EXPECTED RESULT** Path identifies the configured executable; FinishedAt contains a timestamp; docker logs ch2-exit may print nothing because the shell exited without writing output. Empty logs do not mean the container did not run.

2.8 Recognize Isolation Boundaries and Host Dependencies

Container isolation is useful but not absolute. A Linux container relies on the execution environment’s kernel, runtime, permission model, storage, and networking. A compromised or overly privileged container can affect assets beyond itself—especially when given host mounts, host networking, privileged mode, or Docker daemon access.

Five questions to ask before trusting a workload

- Who publishes the image, and where do its layers and dependencies come from?
- What user runs the application, and what Linux privileges does it retain?
- Which host directories, devices, ports, and networks can the container access?
- Can the workload reach the daemon socket or privileged credentials?
- Which storage and recovery guarantees exist after a container is deleted?

Rootless Docker and user-namespace remapping are useful controls, but they require explicit configuration and have operational limitations. Likewise, a Docker Desktop Linux VM adds a separate boundary from the physical host, yet shared directories and other integrations still deserve scrutiny. We will turn these ideas into real controls in Part V.

> **DO NOT RUN THIS** Avoid examples that mount the host root directory or Docker socket into a container merely to “explore” privileges. Those demonstrations introduce disproportionate risk and do not improve the beginner learning goal.

2.9 Compare Docker Desktop and Native Linux Engine Behavior

The same command can run on multiple operating systems, but the daemon’s environment is not identical. On native Linux, a conventional Engine installation runs against the host Linux kernel. Docker Desktop on Windows uses a Linux environment through WSL 2 for Linux containers; Docker Desktop on macOS uses a Linux VM. Docker Desktop for Linux also uses a VM and commonly a distinct desktop-linux context.

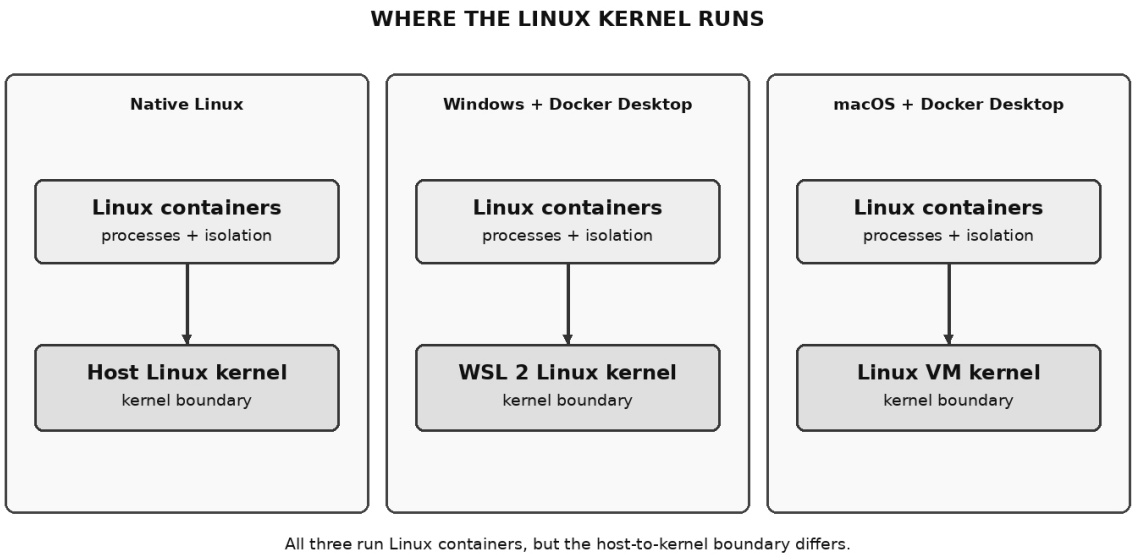


Figure 2.3 — The kernel boundary differs across native Linux Engine and Docker Desktop, even for the same Linux container image.

Environment	Where Linux containers run	What to watch
Windows / Desktop	WSL 2 Linux environment	WSL integration; Windows vs Linux paths
macOS / Desktop	Managed Linux VM	Architecture and file-sharing behavior
Linux / Engine	Host Linux kernel	Daemon permissions; host namespaces
Linux / Desktop	Managed Linux VM	Separate context and image/container store

```
docker context show
docker context ls
docker info --format
'{{.OperatingSystem}}'
```

If an image or container seems to have vanished after switching environments, first check the selected Docker context. It may exist on a different daemon rather than having been deleted. A path such as C:\work\demo, /Users/me/demo, or /home/me/demo can also behave differently when bind-mounted; we cover portability and permissions in Chapters 10 and 12.

2.10 Read a Container's Configuration Before Running It

Imagine a teammate sends you a command and asks you to run an unfamiliar image. The technically mature response is not “it starts, so it must be safe.” Inspect the image source, expected command, default user, architecture, and any runtime privileges or mounts before you grant it resources.

A safe image preflight

```
docker image inspect alpine:3.22 --format '{{.Os}}/{{.Architecture}}'
docker image inspect alpine:3.22 --format '{{json
.Config.Entrypoint}}'
docker image inspect alpine:3.22 --format '{{json .Config.Cmd}}'
docker image inspect alpine:3.22 --format '{{.Config.User}}'
```

You are inspecting package metadata here, not asserting that the image is vulnerability-free. A scanned image may still contain vulnerabilities, and a benign-looking image can be run with dangerous flags. Runtime configuration and image provenance must be considered together.

Controlled execution: deny networking and persistent writes

```
docker run --rm --network none --read-only --cap-drop ALL alpine:3.22 id
```

> **EXPECTED RESULT** You should see a user identity line. This demonstration uses a trusted official test image, no published ports, no host mounts, no network connectivity, a read-only root filesystem, and dropped Linux capabilities. It is a constrained example—not a proof of comprehensive sandbox security.

If the command fails because your execution environment does not support a particular hardening flag, record the exact message and use the supported platform configuration rather than disabling all restrictions indiscriminately. In Part V, you will test least-privilege controls against DeployBoard’s real application behavior.

2.11 Checkpoint: Explain and Inspect the System You Created

Your deliverable is a short inspection record, not merely a list of commands you copied. Work through the following checks on your own Docker Engine. Each answer should be supported by output from a real command.

Evidence to collect	Command or observation	Pass condition
Daemon reachable	docker version	Both client and server appear
Correct context	docker context show	The intended environment is active
Image is present	docker image ls alpine	A 3.22-tagged image appears
Container state	docker inspect ch2-workspace	Recorded state is exited
Same image source	Compare container .Image with image .Id	Identifiers match on this daemon
Privileges	Inspect .HostConfig.Privileged	The lab container is not privileged
Process outcome	Inspect ch2-exit exit code	The status is 7
Ephemeral file	Run the file-absence test	A new container sees no previous file

One final comparison

```
docker image inspect alpine:3.22 --format '{{.Id}}'
docker container inspect ch2-workspace --format '{{.Image}}'
docker container inspect ch2-exit --format
'{{.State.ExitCode}}'
```

The first two outputs should be the same local image ID on your current daemon; the third should be 7. If a command instead reports No such container, verify that you used the exact lab names and have not switched contexts. Recreate only the missing disposable container, rather than deleting unrelated resources.

Clean up only this chapter’s containers

```
docker container rm ch2-workspace ch2-
exit
docker container ls -a --filter name=ch2-
docker image ls alpine
```

> **EXPECTED RESULT** The named lab containers disappear from the filtered listing; the Alpine image remains available. Removing the image is optional, and keeping it avoids another download in later experiments.

You now have a working explanation for what Docker creates, how the CLI reaches the Engine, how images and containers differ, and what to inspect before running a workload. Chapter 3 builds on that understanding: you will

become faster at naming, inspecting, controlling, and interrogating containers without guessing.

If your checkpoint does not pass

Symptom	First useful investigation
Client works; server unavailable	docker context ls; Docker Desktop or dockerd service status
Pull fails	Network, registry reachability, authentication, proxy settings
Container name already exists	docker container ls -a --filter name=ch2-
Image and container IDs differ	Verify image tag, active context, and container image reference
Read-only demonstration fails	Read complete error; check flag support and runtime policy
Logs are blank	Check .State.ExitCode; this lab intentionally emits no logs

Chapter 3 — Work Confidently with the Docker Command Line

Your terminal is now a control panel. Learn how to identify exactly what Docker will change, observe what happened, and recover when the result is not what you intended.

In Chapter 2, you separated Docker images from running containers and learned why the daemon matters. Now you will operate Docker deliberately. Rather than memorizing dozens of flags, you will practice a repeatable skill: identify the object, choose a command, read the evidence, and make the next change only when the evidence supports it.

The chapter workbench

You will work with a tiny Alpine Linux utility container and a local-only Nginx web server. This is an operator exercise, not a new DeployBoard application release. The container names begin with ch3-, so you can distinguish this lab from your other work. Follow the commands in sequence; each exercise expects the earlier containers to exist.

Preflight — run from PowerShell, macOS Terminal, or a Linux shell

```
docker version
docker context show
docker pull alpine:3.22
docker pull nginx:stable-
alpine
```

EXPECTED RESULT The Docker client can contact a Linux-container daemon, the active context is identified, and both pulls finish. The exact digest, download progress, architecture, and version strings will differ.

SCOPE AND SAFETY The commands assume a trusted local Docker installation and network access to Docker Hub. Do not paste sample commands into a production host without checking docker context show first. No broad prune or system-wide cleanup commands are needed.

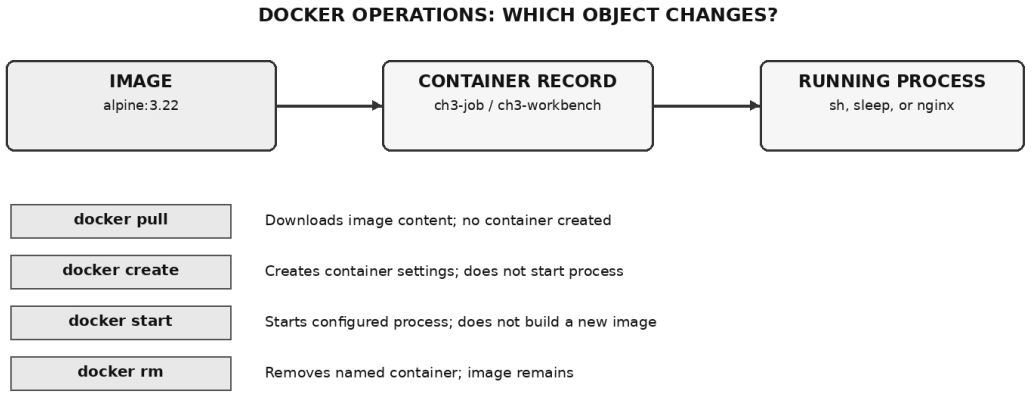


Figure 3.1 — Commands affect different objects: a pulled image, a configured container, and its running process.

3.1 Find and Pull Images from Registries

A registry lets you obtain image content without building it yourself. The image reference you type determines the repository, tag, and, when specified, registry. Docker searches Docker Hub by default for a short name such as alpine:3.22; an explicit registry-qualified name points somewhere else. The downloaded image becomes a local object, but pulling does not start any container.

```
docker image ls
docker image ls alpine
docker pull alpine:3.22
docker image inspect alpine:3.22 --format
'{{.Os}}/{{.Architecture}}'
```

EXPECTED RESULT Alpine appears in your local image list. The inspection output identifies its selected OS and architecture, commonly linux/amd64 or linux/arm64. Images may have different platform variants without changing the tag you typed.

For discovery, Docker Hub's official-image listings and vendor documentation are more reliable starting points than popularity alone. The optional docker search alpine command returns registry search results, not a security endorsement. Check maintainers, image documentation, supported versions, and update policies before choosing an image for production.

What a tag actually guarantees

A tag is a convenient name, not a permanent checksum. Re-running docker pull for a tag can retrieve newly published content. In contrast, a digest reference identifies an exact manifest. You will learn digest pinning and controlled rebuilds in Chapter 8; for this operator lab, the version-family tag keeps commands approachable.

```
docker image inspect alpine:3.22 --format '{{.Id}}'
docker image inspect alpine:3.22 --format '{{json
.RepoDigests}}'
```

INTERPRETATION The image ID and repository digest are not necessarily the same hash. A multi-platform reference can be resolved to a platform-specific image. Do not invent or copy a digest from an example into a production build.

3.2 Create, Name, Start, Stop, and Restart Containers

A container name makes an operation auditable. Compare ch3-job, which runs one short task, with ch3-workbench, which stays alive so you can inspect and modify its working environment. Creating a container prepares its configuration; starting it launches its configured main process.

```
docker create --name ch3-job alpine:3.22 echo job-
complete
docker ps -a --filter name=ch3-job
docker start -a ch3-job
docker inspect -f '{{.State.ExitCode}}' ch3-job
```

EXPECTED RESULT Before startup the job is Created. Attached startup prints job-complete. The job then exits successfully and its recorded ExitCode is 0. It remains listed by docker ps -a until you remove it.

Now create a long-running workspace. The trailing command overrides the image's default command for this particular container; it does not modify the image.

```
docker run -d --name ch3-workbench alpine:3.22 sleep
3600
docker ps --filter name=ch3-workbench
docker stop ch3-workbench
docker start ch3-workbench
docker restart ch3-workbench
docker ps --filter name=ch3-workbench
```

EXPECTED RESULT The workbench appears as Up after start and restart. If you work longer than the one-hour sleep interval, recreate only this named container with a longer duration. Restarting preserves this container's writable state; removing and recreating it does not.

Command	What changes	What does not change
pull	Local image content	No container instance starts
create	New container configuration	The process does not run
start	Existing container process	The image is not rebuilt
stop	Running process becomes stopped	Container record remains
restart	Stops and starts main process	Container name and config remain
rm	Removes a stopped container record	Local image usually remains

3.3 Choose Foreground, Detached, and Interactive Execution

Foreground, detached, and interactive modes solve different operator needs. A short command should ordinarily finish in the foreground so you can see its result. A service normally runs detached. Interactive mode is appropriate for investigating a container, not for making changes that should become part of a repeatable image build.

Compare foreground output with an interactive shell

```
docker run --rm alpine:3.22 echo foreground-
complete
docker ps --filter name=ch3-workbench
docker run --rm -it alpine:3.22 sh
```

At the / # prompt in the interactive container, enter pwd, then exit. The --rm flag removes this temporary container when its main process exits; it does not remove alpine:3.22. In detached mode, -d returns a container ID immediately while the process continues in the background.

TERMINAL REQUIREMENT The -it command needs an interactive terminal. Run it in a normal PowerShell, macOS Terminal, or Linux terminal window. Some notebook consoles and automated CI runners do not provide a TTY; for scripts, omit -t unless the program truly needs one.

Decide before typing

You need to…	Prefer	Why
Get a single result	Foreground; optional --rm	Immediate output and process exit status
Keep a service running	-d with a descriptive --name	Terminal returns while service continues
Inspect a shell manually	-it plus an actual shell command	Human interaction with a temporary session
Attach to an existing service	docker logs / docker exec	Avoid spawning a duplicate service

3.4 Pass Arguments, Commands, and Environment Variables

The location of arguments determines who interprets them. Options before IMAGE customize Docker's container creation; tokens after IMAGE are interpreted as the container command and its arguments. An environment variable passed with -e is available inside the container process; it is not automatically visible to other containers or safe for secrets.

```
docker run --rm -e LAB_MODE=practice alpine:3.22 printenv
LAB_MODE
docker run --rm alpine:3.22 uname -s
docker inspect -f '{{json .Config.Cmd}}' ch3-workbench
```

EXPECTED RESULT The first command prints practice. The second normally prints Linux. The third shows the configured sleep command for ch3-workbench. Container runtime configuration and image defaults are different layers of choice.

Distinguish build-time and run-time data

In Chapter 5, Dockerfile instructions will establish image defaults. For now, treat -e as a runtime setting for non-sensitive configuration, such as an environment name. Avoid placing passwords or API keys directly on command lines; command history and inspection output can reveal them. Chapter 19 teaches appropriate secret handling.

```
docker run --rm --workdir /tmp alpine:3.22
pwd
docker run --rm --user 1000:1000 alpine:3.22
id
```

EXPECTED RESULT The first command prints /tmp. The second identifies UID and GID 1000 in the container; a numerical user is not proof that all files or directories are writable by that user. Always test the application's actual permissions.

3.5 Inspect Logs, Status, Events, and Processes

Operators need evidence at several levels. docker ps gives a fast status snapshot, docker inspect exposes configuration and state, docker logs captures available container output, docker top shows running processes, and docker events records daemon-level events. They answer different questions; none should be treated as a complete monitoring solution.

```
docker ps -a --filter name=ch3-
docker logs --timestamps ch3-job
docker inspect -f '{{.State.Status}}' ch3-
workbench
docker top ch3-workbench
```

> **EXPECTED RESULT** You should find the stopped job and the running workbench. The job log contains job-complete; the workbench may have no application log output because sleep is quiet. Process listings vary by host and daemon configuration.

Use the event history only after the named containers have been created. Recent events can include create, start, stop, and restart. Docker does not promise unlimited event retention, so absence from a past interval is not proof the action never occurred.

```
docker events --since 30m --until 1s --filter type=container --filter container=ch3-workbench
```

> **EXPECTED RESULT** You may see recent workbench events with timestamps and action names. If the lab began more than 30 minutes ago, increase the interval; if the command reports no events, inspect current State and logs instead.

Read a compact status view

```
docker ps -a --format "table
{{.Names}}\t{{.Status}}\t{{.Image}}"
docker inspect -f '{{.State.StartedAt}}' ch3-workbench
```

Logs only report what is made available through the configured logging system. If an application writes messages exclusively to files inside its container, docker logs might be empty. That is an application logging design issue, not evidence the process never ran.

3.6 Execute Commands Inside a Running Container

docker exec starts an additional process inside an already running container. It does not rebuild the image, replace the container's original PID 1, or restart that process. This is ideal for inspecting a running system. It is a poor substitute for putting permanent configuration into a Dockerfile.

```
docker exec ch3-workbench pwd
docker exec ch3-workbench ls
/etc
docker exec ch3-workbench id
docker exec -it ch3-workbench sh
```

At the shell prompt, run ls /tmp and exit. The original sleep process remains running after you leave this diagnostic shell. Try stopping the workbench and running docker exec again; Docker should refuse because exec requires a running container. Start it again before the next section.

```
docker stop ch3-workbench
docker exec ch3-workbench
pwd
docker start ch3-workbench
```

EXPECTED RESULT The middle command fails because the container is stopped. This failure is intentional; record the error and verify the final start restores the running state.

Avoid a common syntax trap

This works: docker exec CONTAINER sh -c "echo one && echo two". Docker exec needs an executable as its command. Passing an entire shell expression as though it were a single executable does not work. For portability, use a shell only when the operation requires shell interpretation.

3.7 Copy Files Between Host and Containers

docker cp is a focused transfer operation, not a persistence strategy. You can copy files into and out of a running or stopped container. The copied file becomes part of the container's writable layer, so removing the container removes that copy. Durable application data belongs in volumes, introduced in Chapter 10.

Prepare a safe working folder

Create a folder for the two text files used by this lab. Choose the block for your terminal, then stay in that folder for the remaining copy commands.

macOS or Linux (Bash or zsh)

```
mkdir -p docker-ch3-lab
cd docker-ch3-lab
printf "operator verified\n" > ch3-
note.txt
```

Windows PowerShell

```
New-Item -ItemType Directory -Force docker-ch3-lab | Out-
Null
Set-Location docker-ch3-lab
Set-Content -Path ch3-note.txt -Value 'operator verified'
```

```
docker cp ./ch3-note.txt ch3-workbench:/tmp/operator-note.txt
docker exec ch3-workbench cat /tmp/operator-note.txt
docker cp ch3-workbench:/etc/alpine-release ./ch3-alpine-
release.txt
```

EXPECTED RESULT The container prints operator verified, and a new local file named ch3-alpine-release.txt appears in your chapter folder. The Alpine release string varies. On Windows, use Get-Content ./ch3-alpine-release.txt; on macOS/Linux, use cat ./ch3-alpine-release.txt.

A successful copy says only that the transfer succeeded. It does not mean the file is backed up, version-controlled, or automatically rebuilt into an image. If you delete ch3-workbench, its /tmp/operator-note.txt copy disappears, while the file in your local chapter folder remains.

PERMISSIONS Copied files may acquire ownership or permissions that differ from the source system. Do not assume that moving a secret into a container makes it secret or that an application user can modify every copied file.

3.8 Set Working Directories, Users, and Resource Options

Container defaults are convenient for experimentation, but professional operators must inspect user identity, working directory, and CPU/memory constraints. Docker flags can enforce limits and establish runtime settings without modifying an image. Limits must fit the host and the workload; an unrealistically low limit can create failures rather than improve reliability.

```
docker run -d --name ch3-limits --workdir /tmp --user 1000:1000 --memory 128m --cpus 0.5 alpine:3.22 sleep 3600
docker exec ch3-limits pwd
docker exec ch3-limits id
docker inspect -f '{{.HostConfig.Memory}}' ch3-limits
docker inspect -f '{{.HostConfig.NanoCpus}}' ch3-limits
```

EXPECTED RESULT The working directory is /tmp, and the user IDs are 1000. Memory is reported in bytes (134217728 for 128 MiB), and the CPU allocation is recorded as 500000000 NanoCPUs. The underlying host may impose additional limits.

```
docker stats --no-stream ch3-limits
docker inspect -f '{{.Config.User}}' ch3-limits
docker inspect -f '{{.Config.WorkingDir}}' ch3-limits
```

Resource statistics are measurements, not guarantees of future demand. Docker Desktop also runs the Linux engine inside a managed environment whose allocated resources may cap container availability. When operating a real application, verify its behavior under load rather than relying on a single stats sample.

Practice a deliberate failure diagnosis

If you see a permission-denied error when a non-root container writes to a directory, first inspect the effective UID, file ownership, and mount permissions. Do not immediately rerun the container as root. You will establish proper application identities and writable paths in Chapters 6 and 18.

3.9 Understand Port Publishing and Loopback Binding

A process can listen on a port inside a container without being reachable from a browser on your host. Port publishing explicitly maps a host address and port to a container port. You already used this in Chapter 1; here you will inspect the exact mapping and explain its security consequences.

```
docker run -d --name ch3-web -p 127.0.0.1:8087:80 nginx:stable-alpine
docker ps --filter name=ch3-web
docker port ch3-web
```

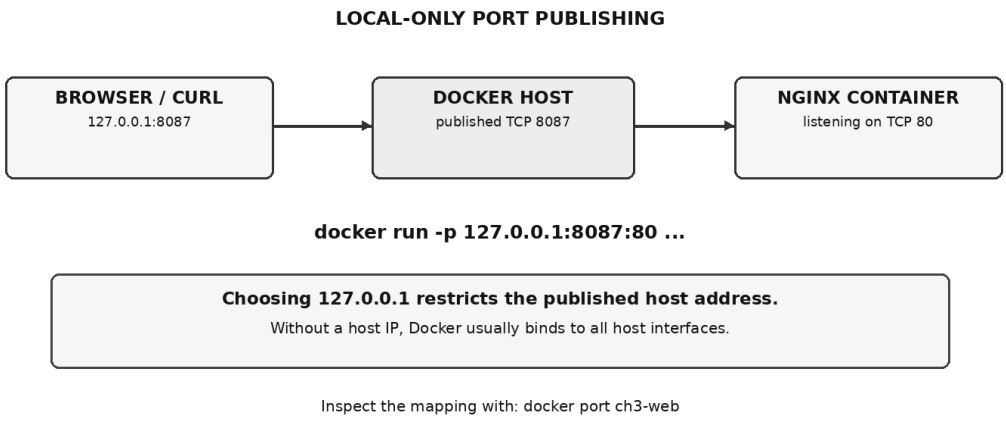


Figure 3.2 — Host loopback 127.0.0.1:8087 is mapped to port 80 inside the Nginx container.

Visit http://127.0.0.1:8087 in your browser. You should see the default Nginx welcome page. An independent terminal check is available below; use the command for your shell.

macOS / Linux (curl)

```
curl -I http://127.0.0.1:8087/
```

Windows PowerShell (curl.exe)

```
curl.exe -I http://127.0.0.1:8087/
```

> **EXPECTED RESULT** The response normally includes HTTP/1.1 200 OK or another successful 2xx status. A different server header or date is not a failure. docker port should report 80/tcp mapped to 127.0.0.1:8087.

Compare -p 127.0.0.1:8087:80 with -p 8087:80. The latter normally publishes on all host interfaces, exposing the service more broadly. Publishing a database or administration interface this way without authorization controls is a serious mistake. Docker Engine releases older than 28.0.0 have a documented edge case in which localhost-published ports can still be reachable from peers on the same local network; update and review your host network controls.

> **TROUBLESHOOTING** If Docker reports that port 8087 is already allocated, stop and remove only ch3-web (if it exists), then try a different unused host port, such as 8097, updating the browser URL accordingly. If the page does not open, check docker ps, docker port ch3-web, and docker logs ch3-web in that order.

3.10 Use Docker Contexts and Command Help Safely

A Docker context determines the daemon that receives your command. This is easy to forget after you start working with remote servers. A harmless-looking docker rm becomes dangerous when the active context points at a production host. A reliable operator reads the context before changing resources.

```
docker context show
docker context ls
docker --help
docker container --
help
docker run --help
```

EXPECTED RESULT docker context show prints the active context name, context ls identifies it with an asterisk, and the help commands describe recognized syntax and options. On Docker Desktop, a context such as desktop-linux is common; do not assume every environment uses default.

Use --context NAME on one command when you need a different target without permanently switching the default. docker context use NAME changes the saved default and can affect later shell sessions; avoid changing it during this introductory lab. Also inspect DOCKER_HOST and DOCKER_CONTEXT if they are configured in your terminal because environment settings may override your expectations.

An operator's three questions

- Which engine will receive this command? Confirm with docker context show and docker info.
- What exact object will change? Prefer a named container and a filtered list before removing anything.
- What can I verify afterward? Check logs, state, a published port, or a file—not just a successful-looking command line.

SECURITY BOUNDARY A remote Docker daemon is a privileged infrastructure control plane. Never publish a raw, unauthenticated Docker API or assume a Docker context is simply a harmless profile. Protected remote access and credentials are addressed later in the book.

3.11 Checkpoint: Complete the Container Operator Challenge

You are acting as the first operator for a small development team. The team needs you to prove that the right image was pulled, a disposable job completed, a service can be reached only through local loopback, runtime limits are present, and a copied report survived back on the host. Do not use screenshots as a substitute for verifiable command output.

Operator challenge — run these checks without recreating your workbench

1. Confirm the active Docker context and locate the images used in this chapter. Record the selected OS/architecture of Alpine.
2. Show that ch3-job is stopped with an exit code of zero and that its log contains job-complete.
3. Show that ch3-workbench is running; inspect its process and read back the file copied to /tmp/operator-note.txt.
4. Confirm that ch3-limits uses UID 1000, runs from /tmp, and has its configured memory and CPU limits.
5. Show that ch3-web maps host 127.0.0.1:8087 to container port 80. Retrieve the default web page or HTTP headers.
6. Record one potential operator mistake and the command you would use to detect it before causing damage.

Reference evidence commands

```
docker context show
docker image ls alpine
docker inspect -f '{{.State.ExitCode}}' ch3-job
docker logs ch3-job
docker top ch3-workbench
docker exec ch3-workbench cat /tmp/operator-note.txt
docker inspect -f '{{.Config.User}}' ch3-limits
docker inspect -f '{{.HostConfig.Memory}}' ch3-limits
docker inspect -f '{{.HostConfig.NanoCpus}}' ch3-
limits
docker port ch3-web
```

Pass criteria

Evidence	Pass condition
Image and target	Correct image is local; active daemon is recognized
Job	Exited (0), expected job output in logs
Workbench	Running; operator note readable in /tmp
Limits	UID 1000, 128 MiB, 0.5 CPU recorded
Web	Nginx responds; publishing is loopback-only
Judgment	Can explain what run, start, exec, cp, and rm change

If any check fails, do not guess: look first at docker context show, docker ps -a, docker inspect, and docker logs. Those four commands typically reveal whether the problem is targeting, lifecycle, configuration, or process output. Retry only the smallest necessary step.

Clean up the named lab objects

After you have collected the evidence, remove only the containers created by this chapter. The copied host files remain in docker-ch3-lab. Keep the downloaded images for the next chapter unless you have a specific reason to free disk space.

```
docker stop ch3-workbench ch3-limits ch3-web
docker rm ch3-job ch3-workbench ch3-limits ch3-
web
docker ps -a --filter name=ch3-
```

> **EXPECTED RESULT** The last listing contains no chapter-3 lab containers. If a named container was not created because an earlier command failed, Docker may report that the name does not exist; remove only those shown by docker ps -a. Never use docker system prune as routine chapter cleanup.

What you now know

You can read a Docker command as an operational change: an image is pulled, a container is created, a process starts, configuration is inspected, evidence is collected, and only then is the disposable object removed. Chapter 4 examines what happens when that process exits badly, a port conflicts, or shutdown and restart behavior surprise you.

Chapter 4 — Control Container Lifecycles and Diagnose First Failures

A container exits during a demonstration. The image is still present, the port no longer responds, and the last command looked correct. Your job is not to guess at a fix. It is to establish what Docker created, what process ran, why that process stopped, and what action will restore the service without losing data.

By the end of this chapter, you can distinguish lifecycle states, choose the right stop or restart action, interpret exit codes and logs, preserve files deliberately, isolate naming and port conflicts, and recover a broken container from observable evidence.

> **LAB SCOPE** All names in this chapter begin with ch04-. Use a Linux-container Docker environment from Chapters 1–3. Commands are given one per line and can be pasted into either Bash or PowerShell unless marked otherwise. Never run the cleanup commands against unrelated containers.

4.1 Understand Created, Running, Exited, and Restarting States

Docker keeps a container object even when its main process has stopped. A container is therefore not synonymous with a running process: it has an image reference, configuration, filesystem layer, name, and state. The four states you will use most often are created, running, exited, and restarting.

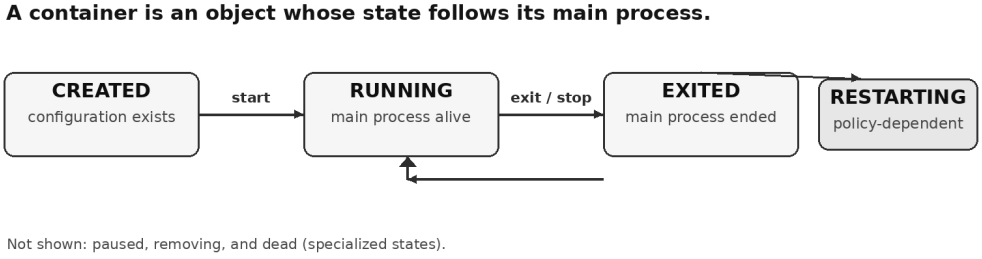


FIGURE 4.1 The working lifecycle: creation, execution, exit, and policy-driven restart.

State	Meaning	Immediate evidence
created	Container exists but its initial process has not started.	docker ps -a shows Created.
running	The initial process is alive.	docker ps shows Up; inspect reports running.
exited	The main process finished, failed, or was stopped.	docker ps -a shows Exited (code).
restarting	Docker is attempting a policy-controlled restart.	docker ps -a may briefly show Restarting.

A running container can also be unhealthy. Health status and lifecycle status are separate signals: an unhealthy process can still be running. Other states such as paused and dead exist, but are not needed for this first incident.

LAB 4.1 — Observe the same container before and after its main process runs

```
docker pull alpine:3.22
docker create --name ch04-state alpine:3.22 sh -c 'echo started; sleep 300'
docker ps -a --filter name=ch04-state
docker start ch04-state
docker ps --filter name=ch04-state
docker stop ch04-state
docker ps -a --filter name=ch04-state
```

Expected progression: Created → Up → Exited. Docker prints a container identifier for create and the container name for start/stop. Your timestamps and IDs will differ. If the 300-second process ends before you stop it, recreate this lab object after removing it.

4.2 Compare docker run, create, start, and exec

These commands operate at different moments. docker create records configuration without launching it. docker start executes the configured initial command on an existing container. docker run is a convenience operation that creates and starts a new container. docker exec launches an additional process inside a container that is already running; it does not replace the initial command.

Command	Changes container identity?	Practical use
docker create	Creates new identity	Prepare a container without starting it.
docker start	Keeps existing identity	Run the saved initial command again.
docker run	Creates new identity	Launch a new instance from an image.
docker exec	Keeps existing identity	Inspect or operate within a running instance.

LAB 4.2 — Run a supplementary command without changing the saved container command

```
docker start ch04-state
docker exec ch04-state sh -c 'echo extra-process; pwd'
docker inspect --format '{{.Config.Cmd}}' ch04-state
docker stop ch04-state
```

The exec process prints its message, but inspect still shows the original shell command and sleep instruction. Exec fails when the container is not running. If you need different initial behavior, create a new container; do not assume docker start accepts a replacement command.

4.3 Configure Restart Policies and Know Their Limits

A restart policy answers a narrow question: should the daemon try to start this container after its main process ends? It does not repair corrupt configuration, restore deleted containers, fix a broken database, or make a single host highly available.

Policy	Daemon behavior	Operator caution
no (default)	No automatic restart.	Appropriate for experiments and one-off jobs.
on-failure[:N]	Restart after nonzero exit, optionally limited to N retries.	Not a substitute for resolving repeated crashes.
always	Restart after exit; special handling after manual stop.	A daemon restart can restart a manually stopped container.
unless-stopped	Restart unless the container was manually stopped.	Still requires a functioning host and daemon.

LAB 4.3 — Observe the meaning of a manual stop under unless-stopped

```
docker run -d --name ch04-restart --restart unless-stopped nginx:alpine
docker inspect --format '{{.HostConfig.RestartPolicy.Name}}' ch04-restart
docker stop ch04-restart
docker ps -a --filter name=ch04-restart
docker start ch04-restart
```

Expected result: the policy reads unless-stopped. After your explicit docker stop, the container stays exited rather than immediately restarting. Starting it manually resumes service. Before changing a real system, distinguish an intentional administrative stop from a crash.

> **IMPORTANT NUANCE** Docker monitors restart behavior only after a container has started successfully (roughly ten seconds). An on-failure container that cannot reach that threshold may not behave like a long-running process that subsequently crashes. Test your actual application rather than relying on a policy name alone.

4.4 Inspect Exit Codes and Common Signals

Every container process exits with a numeric status. Zero conventionally means success; nonzero usually indicates failure, deliberate rejection, or a signal-induced exit. Exit code alone is evidence, not a complete root cause.

LAB 4.4 — Produce a known exit status and inspect it

```
docker run --name ch04-exit alpine:3.22 sh -c 'echo deliberate-failure; exit 7'
docker inspect --format '{{.State.Status}}' ch04-exit
docker inspect --format '{{.State.ExitCode}}' ch04-exit
docker logs ch04-exit
docker wait ch04-exit
```

Expected: exited, 7, and the deliberate-failure log line. docker wait prints the recorded exit code (7); it is not a second execution. The docker run command itself also returns a nonzero status in the terminal for this example.

Observed exit	Common interpretation	Check before concluding
0	Program completed normally.	Was the program meant to stay alive?
1 (or other nonzero)	Application-defined failure.	Logs, args, environment, and dependencies.
125 / 126 / 127	Often Docker invocation / cannot invoke / command not found.	Read the CLI error; distinguish launch failure from app exit.
137	Often terminated by SIGKILL (128 + 9).	Inspect OOMKilled and host evidence; not proof of OOM.
143	Often terminated by SIGTERM (128 + 15).	Check stop history and application signal behavior.

Inspect memory-kill and daemon-level error indicators

```
docker inspect --format '{{.State.OOMKilled}}' ch04-exit
docker inspect --format '{{.State.Error}}' ch04-exit
```

For a normal application exit, State.Error is frequently empty; application failures usually appear in logs. Never label every 137 an out-of-memory condition—an explicit docker kill can produce the same exit status.

4.5 Understand PID 1 and Graceful Shutdown

Inside a Linux container, the initial process is PID 1 in that container's PID namespace. Docker ordinarily requests a graceful shutdown using the configured stop signal (SIGTERM when none is set). If the process does not exit within the stop timeout, Docker sends SIGKILL. Proper signal handling gives an application time to stop accepting work, flush buffers, and release resources.

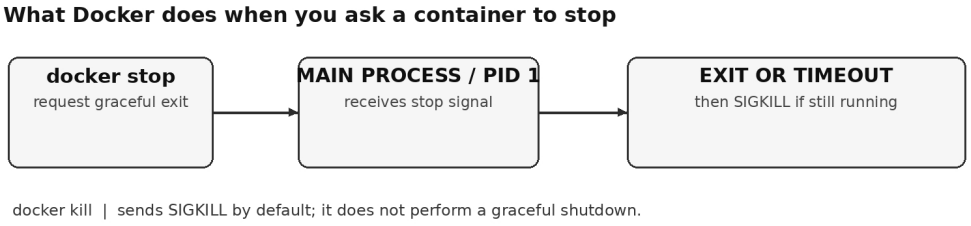


FIGURE 4.2 Graceful shutdown is a timed request, not a promise that the application will comply.

LAB 4.5 — Install a signal handler and watch it shut down

```
docker run -d --name ch04-grace alpine:3.22 sh -c 'trap "echo TERM-caught; exit 0" TERM; while :; do sleep 1; done'
docker top ch04-grace
docker stop --timeout 5 ch04-grace
docker logs ch04-grace
docker inspect --format '{{.State.ExitCode}}' ch04-grace
```

Expected: logs include TERM-caught and the program exits successfully if the trap runs before the timeout. A transient timing variation is possible on slower hosts. A process that ignores termination will instead reach the forced-kill stage.

Why the exec form matters. A Dockerfile command using JSON/exec form launches the application directly, whereas shell-form commands can introduce an intervening shell that may not forward signals properly. Later chapters will use exec-form launches for DeployBoard. Docker also supports the --init option to assist with signal forwarding and orphan-process reaping; --init does not remove the need for application-level shutdown handling.

4.6 Explore Container Filesystems Without Losing Data

A stopped container keeps its writable layer. A removed container does not. This distinction is crucial: stopping and starting the same container is not the same as deleting it and constructing a replacement from the image. Storage and persistent volumes receive full treatment in Chapter 10.

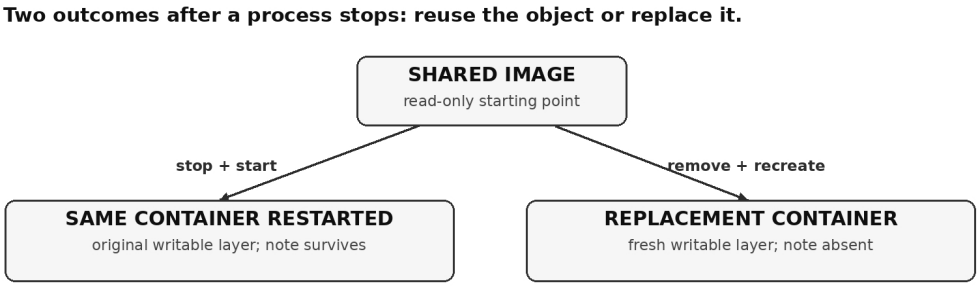


FIGURE 4.3 Writable-layer behavior when an existing container is restarted versus recreated.

LAB 4.6A — Create, inspect, and copy a temporary container file

```
docker run -d --name ch04-fs alpine:3.22 sh -c 'while :; do sleep 1; done'
docker exec ch04-fs sh -c 'echo saved-in-container > /tmp/ch04-note.txt'
docker diff ch04-fs
docker cp ch04-fs:/tmp/ch04-note.txt ./ch04-saved-note.txt
docker stop ch04-fs
docker start ch04-fs
docker exec ch04-fs cat /tmp/ch04-note.txt
```

Expected: docker diff reports an added /tmp/ch04-note.txt (it may report accompanying directory changes); docker cp saves a host-side copy; stop/start retains the in-container note because this is the same container object.

LAB 4.6B — Recreate from the image and prove the temporary data is gone

```
docker rm -f ch04-fs
docker run -d --name ch04-fs alpine:3.22 sh -c 'while :; do sleep 1; done'
docker exec ch04-fs sh -c 'test ! -f /tmp/ch04-note.txt && echo note-is-absent'
```

Expected: note-is-absent. The host copy (./ch04-saved-note.txt) is still present because Docker did not delete your host file. Treat the writable layer as disposable; protect important application state with volumes, backups, and tested recovery procedures.

4.7 Differentiate Stop, Kill, Remove, and Prune

These commands sound similar but have very different consequences. Use the least destructive command that satisfies the operational goal.

Action	Effect	Risk
docker stop NAME	Asks process to end and preserves the container object.	Service downtime; app may require graceful cleanup.
docker kill NAME	Sends SIGKILL by default (configurable).	Interrupts work without normal graceful shutdown.
docker rm NAME	Deletes a stopped container and its writable layer.	Files not stored elsewhere are lost.
docker container prune	Deletes all stopped containers in the selected daemon.	May remove unrelated work; avoid in shared environments.
docker system prune	Cleans multiple categories of unused objects.	Broader irreversible cleanup; inspect before considering.

SAFE PRACTICE — Inspect scope and read cleanup help; do not execute prune here

```
docker ps -a --filter name=ch04-
docker container prune --help
docker system prune --help
```

Do not add --volumes to broad cleanup commands while learning. Containers, images, volumes, and bind-mounted host folders are different data owners; deletion semantics differ. For Chapter 4, remove named lab containers individually at the end.

4.8 Diagnose Port Conflicts and Name Conflicts

A name conflict happens when an object with the requested container name already exists—even if it is stopped. A host-port conflict happens when another process or container already

binds the requested host address and port. Do not confuse host port 8094 with the web server's container port 80.

LAB 4.8A — Publish a test website only to localhost

```
docker run -d --name ch04-web -p 127.0.0.1:8094:80 nginx:alpine
docker ps --filter name=ch04-web
docker port ch04-web
```

Open http://127.0.0.1:8094 and expect the default Nginx welcome page. If the port is already occupied before the lab starts, choose another unused host port consistently. The 127.0.0.1 binding is intended for local access; keep Docker updated and verify exposure on environments with special routing rules.

LAB 4.8B — Trigger the two conflicts intentionally

```
docker create --name ch04-web alpine:3.22 true
docker run -d --name ch04-port-clash -p 127.0.0.1:8094:80 nginx:alpine
docker ps -a --filter name=ch04-port-clash
docker port ch04-web
```

Expected: the first command reports that the name is already in use. The second should fail to bind 127.0.0.1:8094 while ch04-web owns it; Docker may leave ch04-port-clash in Created state. Neither error requires deleting the healthy ch04-web container. Inspect the conflicting object, choose an unused name/port, or stop the owner deliberately.

Remove only the intentionally failed port-conflict container, if Docker created it

```
docker rm -f ch04-port-clash
```

4.9 Trace a Failed Start with Logs and Inspect

When a container exits unexpectedly, separate three questions: Did Docker accept the request? Did the initial process start? What did that process report before it ended? Start with state and logs; inspect configuration only after identifying the missing evidence.

LAB 4.9 — Investigate a container that exits due to missing configuration

```
docker run --name ch04-broken alpine:3.22 sh -c 'if [ -z "$APP_PORT" ]; then echo "FATAL: APP_PORT missing" >&2; exit 64; fi'
docker ps -a --filter name=ch04-broken
docker logs --tail 20 ch04-broken
docker inspect --format '{{.State.Status}} {{.State.ExitCode}}' ch04-broken
docker inspect --format '{{json .Config.Env}}' ch04-broken
```

Expected: the container is exited with code 64, its logs explain APP_PORT is missing, and the saved environment lacks APP_PORT. Do not confuse the status reported by docker ps with a diagnosis. Inspect confirms configuration and lifecycle state; logs provide the application's own explanation.

A repeatable diagnostic decision

If the container is Created, inspect its initial command and any startup error. If it is Exited, read exit code and recent logs. If it is Running but unreachable, check published ports and whether the application listens on the expected interface. If it is Restarting, disable the restart loop for the lab or inspect a stopped replica so you can capture the first failure clearly.

4.10 Recover from a Broken Image or Command

A restart reuses the same immutable container configuration. Restarting a container that has a missing required environment variable will usually repeat the same failure. Recovery means correcting the launch inputs and creating a new instance, not merely retrying the broken one.

LAB 4.10A — Reproduce the failure, then replace the broken configuration

```
docker start ch04-broken
docker logs --tail 5 ch04-broken
docker rm ch04-broken
docker run --name ch04-repaired -e APP_PORT=8080 alpine:3.22 sh -c 'if [ -z "$APP_PORT" ]; then echo "FATAL: APP_PORT missing" >&2; exit 64; fi; echo "ready on port $APP_PORT"'
docker inspect --format '{{.State.ExitCode}}' ch04-repaired
```

Expected: docker start produces the same configuration failure. The corrected new instance prints ready on port 8080 and exits 0 because it is a short-lived verification command, not a server. Replacing the container did not modify the Alpine image.

LAB 4.10B — Differentiate a missing executable from application error output

```
docker run --name ch04-bad-command alpine:3.22 /not-a-real-program
docker ps -a --filter name=ch04-bad-command
docker inspect --format '{{.State.Error}}' ch04-bad-command
docker rm -f ch04-bad-command
docker run --name ch04-good-command alpine:3.22 sh -c 'echo command-repaired'
```

Expected: the first launch fails because the executable does not exist; the exact error message and retained object state can vary with engine/runtime behavior. The replacement succeeds. If an image itself cannot be found, confirm its repository and tag with docker image ls and docker pull before rebuilding or recreating. Never silently replace a production image with an unverified latest tag.

4.11 Checkpoint: Fix a Deliberately Broken Container

You are on duty for a local preview service. A teammate created the object, but it cannot serve the page. Your task is to diagnose it from evidence, replace the faulty launch configuration, and demonstrate a working web response. Do not skip directly to deletion.

The incident fixture

Create the intentionally broken preview container

```
docker create --name ch04-case -p 127.0.0.1:8095:80 nginx:alpine sh -c 'echo "fatal: wrong launch command" >&2; exit 42'
docker start ch04-case
docker ps -a --filter name=ch04-case
```

Your operator brief

Record (1) the container state, (2) its exit code, (3) the log message, (4) the configured startup command, and (5) why docker start alone cannot correct it. Then create a working replacement without exposing the test web server beyond loopback.

Reference recovery sequence — use after you have documented the failure

```
docker logs ch04-case
docker inspect --format '{{.State.Status}} {{.State.ExitCode}}' ch04-case
docker inspect --format '{{json .Config.Cmd}}' ch04-case
docker rm ch04-case
docker run -d --name ch04-case -p 127.0.0.1:8095:80 nginx:alpine
docker ps --filter name=ch04-case
docker port ch04-case
```

Expected: the broken run reports exit 42 and the diagnostic message. After replacement, the container is Up, its published endpoint is 127.0.0.1:8095, and opening http://127.0.0.1:8095 displays the Nginx welcome page. If the host port is occupied, move to a known unused localhost port and record your change.

Evidence required to pass

Proof	Passing observation
State identified	Exited before repair; Running after replacement.
Failure explained	Startup command deliberately exited 42 with a fatal message.
Correct correction	New container uses the image's intended default command.
Exposure controlled	Only localhost host-port binding was published.
Browser verification	Nginx welcome page responds on the chosen local port.

Clean lab objects carefully

Only remove the names you created in this chapter. Some containers may already have been removed by an earlier lab; skip any that do not exist. The copied host file is separate from Docker and may be deleted manually once you have inspected it.

CHAPTER CLEANUP — Run after the checkpoint, ignoring only "No such container" for optional labs

```
docker rm -f ch04-state ch04-exit ch04-restart ch04-grace
docker rm -f ch04-fs ch04-web ch04-repaired ch04-good-command
docker rm -f ch04-case
docker ps -a --filter name=ch04-
```

Delete ./ch04-saved-note.txt from your working folder only if it is your lab copy. We intentionally do not use prune, remove images, or delete any named volumes.

What you can now do. You can diagnose whether Docker failed to create an object, a container process failed to run, or a healthy service could not be reached. You can use direct evidence to decide whether to start, stop, replace, or preserve a container. Chapter 5 turns that operating discipline into your first purpose-built Docker image.

PART II

CREATE SMALL, REPRODUCIBLE DOCKER IMAGES

Chapter 5 — Write Your First Dockerfile and Package a Website

Make a website once, package it as an image, and prove the same content can run without borrowing files from your laptop.

YOU WILL PRODUCE	YOU WILL PROVE
An nginx-based website image named deployboard-web:1.0	The container serves files copied into the image, not files mounted from your host.
A rebuilt image named deployboard-web:1.1	Changing your source requires a rebuild and a replacement container.

Starting point: Docker Engine is running, docker commands work, and you can open a local browser. Chapters 1–4 established those skills. No Docker Hub account, domain, paid hosting, or Git repository is needed for this chapter.

5.1 Set Up the First Project Directory

In Chapter 1, you ran an existing image. Now you become the image publisher. The input is a plain directory on your computer; the deliverable is a reusable Docker image. Keeping the input small makes every later inspection meaningful.

Create the workspace

Open a terminal in a location where you normally keep learning projects. Use the two commands that match your shell, then open the new folder in a text editor.

Listing | Windows PowerShell, macOS Terminal, or Linux shell

```
mkdir deployboard-web
cd deployboard-web
```

Inside deployboard-web, create a folder named site and three text files exactly as shown. Make sure the Dockerfile has no file extension; Windows Explorer sometimes hides extensions.

Listing | Project tree after setup

```
deployboard-web/
|-- Dockerfile
|-- .dockerignore
`-- site/
   |-- index.html
   `-- styles.css
```

If you create the files in an editor, save index.html and styles.css as plain UTF-8 text. Use lowercase names. Docker copies filenames literally, and filesystems differ in case sensitivity.

Why the directory matters. The final dot in docker build -t ... . will select this folder as the build context. Files outside it are not automatically available to COPY; Chapter 7 will examine more advanced build contexts.

5.2 Write a Simple Static Website from Scratch

You will publish a release card for DeployBoard, the application that becomes more capable later in the book. At this stage it is deliberately static: no Python service, database, or API is involved. You are learning the boundary between source files and an image, not adding unnecessary dependencies.

File: site/index.html — complete listing

Listing | 5.1 — HTML source (copy into site/index.html)

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <title>DeployBoard | Release Card</title>
 <link rel="stylesheet" href="styles.css">
</head>
<body>
 <main class="frame">
  <p class="eyebrow">DEPLOYBOARD / RELEASE CARD</p>
  <header class="intro">
   <h1>Built once. Ready to ship.</h1>
   <p>A small website packaged inside a Docker image.</p>
  </header>
  <section class="panel" aria-labelledby="snapshot-title">
   <h2 id="snapshot-title">Release snapshot</h2>
   <dl class="facts">
    <div><dt>Artifact</dt><dd>deployboard-web:1.0</dd></div>
    <div><dt>Runtime</dt><dd>NGINX web server</dd></div>
    <div><dt>Delivery</dt><dd>Docker image</dd></div>
   </dl>
  </section>
  <p class="closing">No host bind mount required.</p>
 </main>
</body>
</html>
```

File: site/styles.css — complete listing

Listing | 5.2 — CSS source (copy into site/styles.css)

```
* { box-sizing: border-box; }
html { font-family: Arial, Helvetica, sans-serif; }
body {
 margin: 0;
 background: #eef1f3;
 color: #20252a;
 line-height: 1.5;
}
.frame {
 max-width: 740px;
 margin: 9vh auto;
 padding: 36px;
}
.eyebrow {
 font-size: 12px;
 font-weight: bold;
 letter-spacing: 0.12em;
}
.intro h1 {
 margin: 12px 0 8px;
 font-size: clamp(28px, 5vw, 46px);
 line-height: 1.12;
}
.intro p { margin-top: 0; color: #485159; }
.panel {
 margin-top: 28px;
 padding: 24px;
 background: #fff;
```

```
  border: 2px solid #262c32;
  border-radius: 12px;
}
.panel h2 { margin: 0 0 12px; font-size: 20px; }
.facts { margin: 0; }
.facts div {
  display: flex;
  justify-content: space-between;
  gap: 16px;
  padding: 10px 0;
  border-top: 1px solid #d9dde0;
}
.facts dt { font-weight: bold; }
.facts dd { margin: 0; text-align: right; }
.closing { margin-top: 22px; font-weight: bold; }
@media (max-width: 540px) {
  .frame { margin: 0; padding: 24px; }
  .facts div { display: block; }
  .facts dd { text-align: left; }
}
```

These are the entire website files. The HTML links to styles.css with a relative path, so both files must be copied into the same web root. The status details are sample text; they are not generated dynamically from Docker metadata.

Quick source check. In your editor, confirm that the HTML contains “Built once. Ready to ship.” and that the stylesheet contains .panel. If the files are accidentally saved as index.html.txt or styles.css.txt, the site will not render as intended.

5.3 Select and Verify a Suitable Base Image

A Dockerfile starts from an image that already supplies an operating-system userspace and the software your application needs. Since this project serves static HTML, the official NGINX image is a better starting point than installing a web server manually inside a general-purpose Linux image.

Use nginx:stable-alpine for the learning build. It is a maintained, moving tag in the Docker Official Images catalog—not a fixed release identifier. This choice keeps the first project short. Chapter 8 covers digest pinning, precise image identity, and managed updates.

Listing | Pull and inspect the selected base image

```
docker pull nginx:stable-alpine
docker image inspect nginx:stable-alpine --format "{{.Os}}/{{.Architecture}}"
docker image inspect nginx:stable-alpine --format "{{.Id}}"
```

Expect linux/amd64, linux/arm64, or another platform matching the image selected by Docker, followed by an image ID beginning with sha256:. Your exact ID will differ and can change as the upstream tag is updated.

How to judge a base image

QUESTION	DECISION FOR THIS PROJECT
Does it provide the necessary server?	Yes. Official NGINX images include a configured web server.
Is the source documented?	Yes. Docker Hub identifies the official image and its maintainers.
Is the tag immutable?	No. Record its resolved image ID; learn digest pinning in Chapter 8.
Does smaller always mean safer?	No. Size, patching, compatibility, and maintenance all matter.

> **TRUST BOUNDARY** Image names are not proof of trust. Avoid lookalike repositories and inspect the official publisher and tag documentation before building. An image pulled today may not be byte-identical to a pull months later.

5.4 Understand FROM, COPY, WORKDIR, and EXPOSE

Only four Dockerfile instructions are needed for the first image. Their meaning is simple once you keep the build-time and run-time responsibilities separate.

TOKEN	WHAT IT DOES	COMMON MISREADING
FROM	Starts the build from a base image.	It does not start a container.
WORKDIR	Sets the working directory for later relevant instructions.	It does not change your host terminal directory.
COPY	Copies files from the build context into the image.	It is not a live synchronization or mount.
EXPOSE	Documents an intended container port.	It does not publish that port to your host.

The NGINX image serves static content from /usr/share/nginx/html by default, and the bundled NGINX configuration listens on port 80. For this specific image, WORKDIR makes the target explicit and lets COPY use a relative destination. EXPOSE 80 records intent; docker run -p will create the actual host-to-container port mapping.

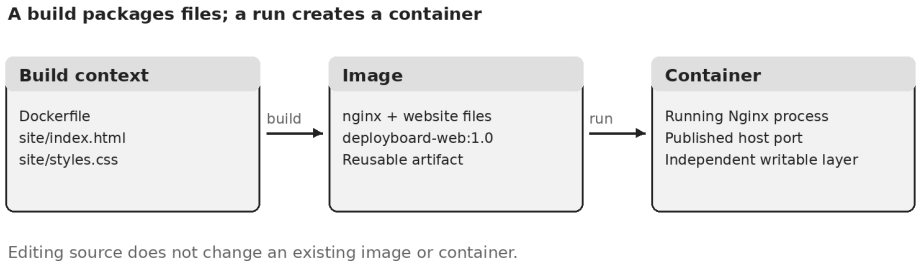


Figure 5.1 — Building produces an image; running the image produces a container.

5.5 Create a Dockerfile for the Website

Create a file named Dockerfile in the deployboard-web folder, beside site rather than inside it. The file is short enough to audit completely.

Listing | 5.3 — Complete Dockerfile

```
FROM nginx:stable-alpine
WORKDIR /usr/share/nginx/html
COPY site/ ./
EXPOSE 80
```

Read it from top to bottom. FROM chooses the NGINX base. WORKDIR selects the web root inside the image. COPY places index.html and styles.css into that directory. EXPOSE documents TCP port 80. You do not need RUN, CMD, or ENTRYPOINT here because the NGINX base image already defines how its server starts.

A trailing slash in site/ emphasizes that the directory contents are the copy source. The ./ destination is the current WORKDIR. Do not write COPY ./ /usr/share/nginx/html: that would also copy unrelated project files if you later added them to the context.

The first design decision

Keep the website separate from the container image instructions. Your source belongs in site; the Dockerfile describes packaging. Chapter 6 will use the same separation when containerizing a Python API, with additional concerns such as dependencies and the application process.

> **PACKAGING RULE** Do not add passwords, tokens, private keys, or .env secrets to this build. Even a removed secret may remain discoverable in image history or layers if it was added during a build. Chapter 19 will demonstrate safe secret handling.

5.6 Build and Tag the Image

From inside deployboard-web, run the build. The tag names a local image artifact. This is not yet a Docker Hub publication.

Listing | Build image version 1.0

```
docker build -t deployboard-web:1.0 .
```

The trailing dot is essential: it tells Docker to send the current folder as the context. The build should complete with a success or export message. Build progress differs by Docker version and builder; do not expect an identical timestamp, image ID, or step count.

Listing | Confirm the image and retrieve its immutable local ID

```
docker image ls deployboard-web
docker image inspect deployboard-web:1.0 --format "{{.Id}}"
```

Expected output | Illustrative result only; IDs, times, sizes vary

```
REPOSITORY       TAG   IMAGE ID      CREATED         SIZE
deployboard-web  1.0   <image-id>    <recent time>   <varies>
```

The image reference deployboard-web:1.0 is convenient, but its tag can be reassigned by a future build. The sha256 image ID identifies the current local image content. Registry digests and multi-platform identity are different concepts, addressed later.

If the build fails. Check your terminal location with pwd (Bash) or Get-Location (PowerShell). Dockerfile must be at the context root, and site/index.html must exist. A failed COPY usually means Docker cannot find the specified path in the build context.

5.7 Run the Website Without a Bind Mount

Now create one named container from the packaged image. Its website files will be read from the image; you will not mount the host project directory.

Listing | Start and check the running website

```
docker run -d --name db-web-v1 -p 127.0.0.1:8086:80 deployboard-web:1.0
docker container ls --filter name=db-web-v1
```

Open http://127.0.0.1:8086 in a browser. You should see “Built once. Ready to ship.” above the release snapshot. The host-side 8086 port is intentionally bound to the IPv4 loopback address so this introductory site is not broadly published on the local network. Docker documents a same-L2-network exception for localhost publishing on Engine releases older than 28.0.0; use a current patched Engine and do not treat this demonstration as a production security boundary.

Listing | Inspect the source image and mounts

```
docker container inspect db-web-v1 --format "{{.Config.Image}}"
docker container inspect db-web-v1 --format "{{json .Mounts}}"
```

The first command should print deployboard-web:1.0. The mounts result should be an empty JSON array ([]) for this particular container; no -v or --mount option was supplied, and this project

declares no Docker volumes. This is an observation about the exercise, not a universal statement about every image.

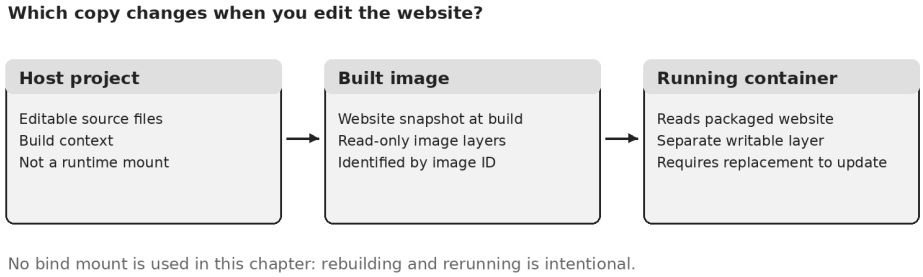


Figure 5.2 — Source files, a built image, and a running container are separate states.

5.8 Compare the Image with Its Running Container

An image is a reusable template with read-only image layers. A container is a runtime instance with its own writable filesystem state. You can inspect both without guessing how the image was built.

Listing | Compare image identity with container state

```
docker image inspect deployboard-web:1.0 --format "{{.Id}}"
docker container inspect db-web-v1 --format "{{.Image}}"
docker container inspect db-web-v1 --format "{{.State.Status}}"
```

The two IDs should match, because the container was created from that exact local image. The final line should be running. The first two commands do not return the image tag: they return the image content identifier recorded on the image and container.

Listing | Inspect files copied into the running container

```
docker container exec db-web-v1 ls -l /usr/share/nginx/html
docker container exec db-web-v1 cat /usr/share/nginx/html/index.html
```

You should see index.html and styles.css; the HTML should include the exact headline from Listing 5.1. NGINX base images may include additional files in the web root, so focus on the two files you supplied. The container is reading copies packaged during the build, not live host files.

A controlled source-change experiment

While db-web-v1 keeps running, open site/index.html on your host and briefly replace “Built once. Ready to ship.” with “Edited on my laptop.” Refresh the browser without rebuilding. The running site should still show the old headline. Undo that temporary edit before proceeding to Section 5.10. This proves the packaging boundary; it is not a browser-cache guessing game.

5.9 Use .dockerignore to Control Build Context

The context is more than the Dockerfile and the files you intend to copy. Without filtering, source-control data, editor artifacts, temporary drafts, or accidentally added credentials may enter the build context. A .dockerignore file prevents matching paths from being available to ordinary context-based COPY instructions.

File: .dockerignore — complete listing

Listing | 5.4 — Build-context exclusions

```
# Editor, operating-system, and temporary files
.git
.DS_Store
```

```
Thumbs.db
*.log
*.tmp

# Local-only data, drafts, and credentials
drafts/
notes/
secrets/
.env
.env.*
```

For this project, COPY site/ ./ is already narrow. .dockerignore adds another guardrail: as the project grows, ignored files do not need to be transferred into the build context. This is not a substitute for keeping secrets out of the project and out of Dockerfile instructions.

Listing | Rebuild to inspect plain-text BuildKit progress

```
docker build --progress=plain -t deployboard-web:1.0 .
```

Look for lines about loading the build context and transferring context. Transfer sizes are implementation-dependent; use them as evidence that context handling exists, not as a performance benchmark. The familiar cached result for unchanged website files is expected on subsequent builds.

One important distinction. .gitignore affects what Git tracks; .dockerignore affects what the Docker builder receives. They are independent controls. For advanced build-cache behavior and file-specific ignores, see Chapter 7.

5.10 Change Source Files and Rebuild Correctly

A published artifact must have a clear update path. You have already observed that a local source edit does not change the running container. Now perform a deliberate versioned update.

Make exactly two edits in site/index.html

Replace the following two complete text values; leave the rest of the HTML untouched:

Listing | Changes required for release 1.1 (not terminal commands)

```
OLD: Built once. Ready to ship.
NEW: Updated and ready to ship.

OLD: <dd>deployboard-web:1.0</dd>
NEW: <dd>deployboard-web:1.1</dd>
```

The page is static, so the displayed release value is edited manually. Docker tags are not automatically injected into HTML.

Listing | Build 1.1 and replace the server on port 8086

```
docker build -t deployboard-web:1.1 .
docker stop db-web-v1
docker rm db-web-v1
docker run -d --name db-web-v2 -p 127.0.0.1:8086:80 deployboard-web:1.1
```

Refresh http://127.0.0.1:8086. You should see “Updated and ready to ship.” and deployboard-web:1.1. Because the old container was removed, the name and host port are available. Docker does not automatically replace a running container when you build a new image.

Prove the previous release still exists

Listing | Start release 1.0 beside release 1.1

```
docker image ls deployboard-web
docker run --rm -d --name db-web-old -p 127.0.0.1:8087:80 deployboard-web:1.0
```

The original release should still be visible at http://127.0.0.1:8087, while release 1.1 appears at http://127.0.0.1:8086. If you inspect both URLs, you will see that one source tree produced two separate versioned image artifacts.

Listing | Stop old comparison container; remove current release container

```
docker stop db-web-old
docker container ls --filter name=db-web-v2
```

The db-web-old container used --rm, so docker stop removes it automatically. Keep db-web-v2 running for the checkpoint below; the final cleanup in Section 5.11 removes it after verification.

5.11 Checkpoint: Ship a Self-Contained Web Image

Your outcome is not merely a page visible in a browser. It is an image with known inputs and proof that the packaged content runs without a host bind mount. Use this evidence record before proceeding to Chapter 6.

Operator acceptance tests

7. The project folder contains Dockerfile, .dockerignore, site/index.html, and site/styles.css; there are no required downloads beyond the NGINX base image.
8. docker image ls deployboard-web lists versions 1.0 and 1.1 after the two builds; each has its own image ID.
9. Release 1.1 serves the revised headline on 127.0.0.1:8086, and release 1.0 can independently serve the previous headline on 127.0.0.1:8087.
10. Inspecting the web container shows its image source and no project-directory bind mount.
11. The NGINX process is running and the static page remains reachable when you leave the editor closed.

Quick inspection commands (run while db-web-v2 is running)

Listing | Prove runtime state, image reference, and packaging

```
docker container ls --filter name=db-web-v2
docker container inspect db-web-v2 --format "{{.Config.Image}}"
docker container inspect db-web-v2 --format "{{json .Mounts}}"
docker image ls deployboard-web
```

Final cleanup — project-scoped and non-destructive

Listing | Remove the running exercise container; retain built images

```
docker stop db-web-v2
docker container ls --filter name=db-web-v2
docker image ls deployboard-web
```

Do not run docker system prune, docker volume prune, or docker image prune as part of this checkpoint. Those commands may affect unrelated projects. Keeping the two named images gives you a reference point for the next chapter, where DeployBoard becomes a real API.

What you can now explain. An image is built from a Dockerfile and context; a container is created from the image; EXPOSE documents a port, whereas -p publishes it; a source edit affects neither an existing image nor its running container until you rebuild and replace deliberately.

> **WHEN A CHECK FAILS** If the first build could not download nginx:stable-alpine, verify Docker connectivity and image pull permissions. If the page is unreachable, check docker container ls, docker logs db-web-v2, and whether local port 8086 is already in use. If the page displays the

> old headline, check the image tag used to create the current container before clearing a browser cache.

Chapter 6 — Containerize a Python API the Right Way

Your static release-card website in Chapter 5 was a complete artifact: the image held its files, and the running container served them. An API has a different job. It must start a long-running process, accept HTTP requests, read configuration, and stop predictably when Docker asks it to. In this chapter, you will package a small service that does those things without introducing a database, a queue, or unnecessary application complexity.

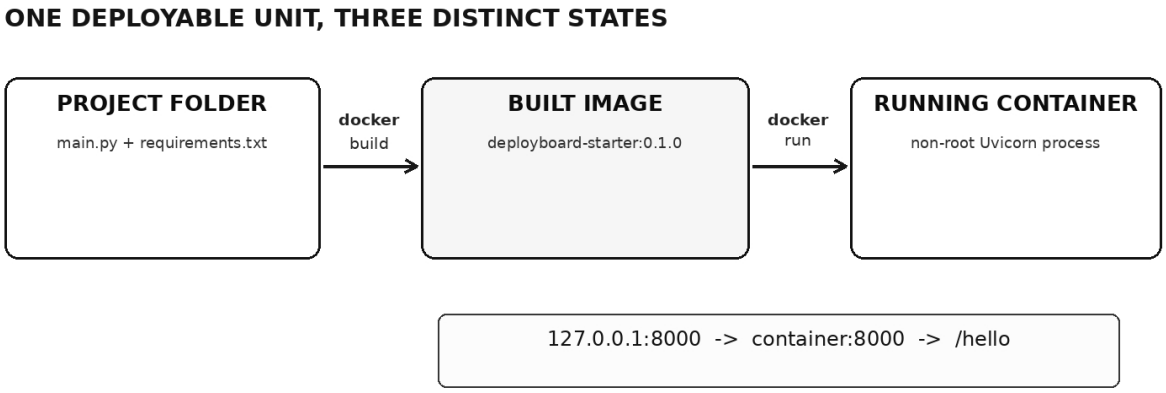


Figure 6.1 — Source, built image, and running container are related but distinct.

The delivery brief

Starting point: Docker works, and you have finished the static website exercise. **End state:** a tagged image named `deployboard-starter:0.1.0` runs FastAPI as a non-root user, answers `/health` and `/hello`, and shuts down cleanly. **Lab scope:** one Linux container on Docker Desktop or Docker Engine, using port 8000 on your own computer.

> **BOUNDARY** This is the deliberately stateless starter for DeployBoard. The real deployment records, CRUD endpoints, authentication, migrations, and database integration begin in Chapter 13. Nothing here requires you to know PostgreSQL.

6.1 Meet DeployBoard Through a Small Stateless Service

DeployBoard will eventually track releases and the health of deployment environments. For now, imagine a teammate asking you to ship only the

tiniest component that can answer a request. You need a service that proves a request crossed the host-to-container boundary and reached Python. It does not need persistent records.

We will make two routes. `/health` returns a fixed status so you can distinguish an answering process from one that failed to start. `/hello` returns a configurable message and the name of the environment. Both responses are small JSON objects. The values do not survive through storage because no state is written; changing the runtime environment only changes how the process responds.

Contract	Request	Response
Health	GET /health	200 JSON: status=ok, service=deployboard-starter
Greeting	GET /hello	200 JSON: message=<configured>, stage=<configured>
Unknown route	GET /missing	404 Not Found

The contract is intentionally modest. A health endpoint is useful evidence that the HTTP process responded; it is **not** proof that a future database or worker is ready. Later chapters will distinguish process health, service readiness, and dependency health.

6.2 Create a Minimal FastAPI Health-and-Hello Application

Make a new working directory called `deployboard-starter`. Place it alongside the Chapter 5 website project rather than inside it. Open this directory in a plain-text editor or IDE and create an `app` subdirectory. File names are case-sensitive inside a Linux container even when your host operating system is not.

MACOS / LINUX TERMINAL

```
mkdir -p deployboard-starter/app
cd deployboard-starter
```

WINDOWS POWERSHELL

```
New-Item -ItemType Directory -Force deployboard-starter/app
Set-Location deployboard-starter
```

Create `app/main.py` with the exact contents below. FastAPI maps Python callables to HTTP routes; Uvicorn, introduced later, is the ASGI server that

listens for requests and calls your app. Do not insert a local development reload option into this program.

FILE: APP/MAIN.PY

```
"""Small, deliberately stateless API used in Chapter 6."""

import os
from fastapi import FastAPI

app = FastAPI(title="DeployBoard Starter", version="0.1.0")

@app.get("/health")
def health() -> dict[str, str]:
    return {"status": "ok", "service": "deployboard-starter"}

@app.get("/hello")
def hello() -> dict[str, str]:
    return {
        "message": os.getenv("APP_GREETING", "Hello from DeployBoard"),
        "stage": os.getenv("APP_STAGE", "local"),
    }
```

Notice that the file contains no database client and no global mutable counter. Each endpoint computes a response from fixed values or explicitly supplied runtime settings. That is enough to test packaging, configuration, and lifecycle behavior without hiding Docker problems behind unrelated application logic.

> **READ THE OUTPUT** When the app runs without environment overrides, `/hello` will report `Hello from DeployBoard` and `local`. Later we will pass different values at container creation time and verify that the image itself did not change.

6.3 Organize Source Files and Dependency Definitions

A good container build starts with a predictable directory. Create `app/__init__.py`, `requirements.txt`, and `.dockerignore` at the indicated paths. This is the complete starter project; there is no supplemental repository and no omitted application module.

FILE: APP/__INIT__.PY

```
"""DeployBoard Starter application package."""
```

FILE: REQUIREMENTS.TXT

```
fastapi==0.146.0
uvicorn==0.53.0
```

These direct package versions are intentionally fixed for the book's October 2026 baseline. FastAPI supplies the web framework; Uvicorn is the ASGI server. Direct pins reduce accidental version drift, but they do not lock transitive dependencies or the image's base digest. Chapter 8 will introduce stronger release reproducibility and exact artifact identity.

FILE: .DOCKERIGNORE

```
.git
.venv
__pycache__
*.pyc
.env
*.pem
*.key
.pytest_cache
.DS_Store
```

Build context is the set of files Docker sends to the builder. Exclude virtual environments, local credentials, caches, and Git metadata before copying source into an image. `.dockerignore` is not an access-control system: do not put secrets in the project tree and assume ignored patterns make them safe.

CANONICAL PROJECT TREE

```
deployboard-starter/
|-- Dockerfile
|-- .dockerignore
|-- requirements.txt
`-- app/
   |-- __init__.py
   `-- main.py
```

VERIFICATION Your working terminal should be inside `deployboard-starter` when you run `docker build`. Run `pwd` on macOS/Linux or `Get-Location` in PowerShell to confirm. An incorrect build directory is a common cause of COPY failures.

6.4 Choose a Python Base Image Deliberately

An application image starts with a base. We will use `python:3.13-slim-bookworm`, a Docker Official Image variant that provides Python on a smaller Debian Bookworm userland. This is a purposeful middle ground: smaller than a general-purpose Python image, but without introducing Alpine/musl compatibility differences in the first Python container lab.

Base choice	Practical consequence
python:3.13-slim-bookworm	Chosen: Debian-based, relatively compact, straightforward for pure-Python packages.
python:3.13-bookworm	Larger general-purpose base; useful when broader OS tooling is essential.
Alpine-based Python variant	May be smaller but musl-based; native binary wheels and debugging differ.
Floating tags such as python:latest	Not suitable for repeatable learning or controlled releases.

The selected tag fixes a Python minor series and Debian distribution but can still receive updated contents over time. That is desirable for some security maintenance workflows and insufficient for bit-for-bit reproducibility. Later, when you publish a release, inspect and record the actual image digest. Do not invent a digest or paste one from a different CPU architecture.

All commands in this chapter build **Linux containers**. On Windows and macOS, Docker Desktop provides a Linux runtime; you do not need a Windows-container base image. Building and running requires a network connection the first time dependencies or base layers are downloaded.

6.5 Write the Application Dockerfile

Now turn the source folder into an image recipe. Create `Dockerfile` (no extension) at the project root and paste the complete listing. We deliberately keep the first build to one stage; Chapter 7 will measure and improve its build performance before introducing multi-stage variants.

FILE: DOCKERFILE — COMPLETE

```
FROM python:3.13-slim-bookworm

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1
WORKDIR /srv/deployboard

COPY requirements.txt ./
```

```
RUN pip install --no-cache-dir --disable-pip-version-check -r requirements.txt

RUN groupadd --gid 10001 appgroup \
    && useradd --uid 10001 --gid appgroup \
       --create-home --shell /usr/sbin/nologin appuser
COPY --chown=appuser:appgroup app/ ./app/

ARG BUILD_VERSION=0.1.0
LABEL org.opencontainers.image.version="${BUILD_VERSION}"
USER appuser
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Read the Dockerfile as a sequence of build decisions. `FROM` selects the Python runtime. The two `ENV` settings prevent Python from leaving bytecode files and keep output unbuffered for container logs. `WORKDIR` gives commands a predictable working directory. Copying `requirements.txt` before the application allows dependency installation to reuse its cached layer when only Python source changes.

The `RUN` instruction installs the declared packages. The next `RUN` creates a dedicated account, and `COPY --chown` makes the application files owned by that account. `USER` changes the runtime identity; it does not claim the whole host is protected against all attacks. `EXPOSE 8000` documents the intended listening port, but **does not publish the port**. `CMD` supplies the default process that will run when a container starts.

> **WHY ROOT APPEARS DURING BUILD** A build step may need elevated privileges to install dependencies or create users. The finished application process does not: it listens on port 8000 and serves read-only Python code. Chapter 18 will harden the application further.

6.6 Set Workdir, Runtime User, and Launch Command

First, build a named image. Docker treats the final period as the build context directory. Use the same command in a Unix shell or PowerShell while positioned in the project root.

BUILD THE STARTER IMAGE

```
docker build --progress=plain -t deployboard-starter:0.1.0 .
```

A normal build shows the base image being resolved, packages installed, files copied, and the result exported or loaded as an image. Exact step counts, IDs, timings, and cache messages vary by Docker version and

architecture. A successful build ends without an error and makes the tag available to `docker image ls`.

CHECK ITS CONFIGURATION

```
docker image ls deployboard-starter
docker image inspect --format "{{.Config.WorkingDir}}" deployboard-starter:0.1.0
docker image inspect --format "{{.Config.User}}" deployboard-starter:0.1.0
```

EXPECTED OUTPUT / EXPECTED VALUES (ILLUSTRATIVE)

```
/srv/deployboard
appuser
```

The first value proves you set the execution directory; the second proves the image declares a non-root runtime account. Neither one guarantees that a running service is reachable. The next section starts it and tests the exposed HTTP behavior.

6.7 Explain ENTRYPOINT, CMD, and exec Form

Container startup has two related controls. `CMD` supplies defaults for the container command or default arguments to an `ENTRYPOINT`. `ENTRYPOINT` defines an executable that normally remains fixed while arguments can vary. This starter deliberately uses an exec-form `CMD` alone so the Uvicorn executable can run as the container's main process.

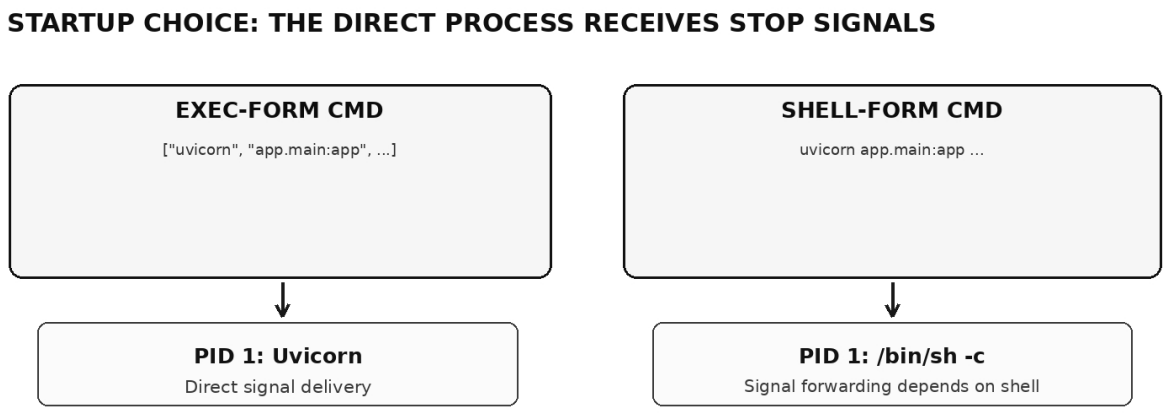


Figure 6.2 — Exec form avoids a default shell wrapper between Docker and Uvicorn.

INSPECT THE IMAGE STARTUP SETTINGS

```
docker image inspect --format "{{json .Config.Cmd}}" deployboard-starter:0.1.0
docker image inspect --format "{{json .Config.Entrypoint}}" deployboard-starter:0.1.0
```

The `Cmd` output should contain the Uvicorn JSON-array command. The `Entrypoint` value depends on any setting inherited from the base image;

inspect it instead of guessing. We did not define a new `ENTRYPOINT` in this Dockerfile. The exec form keeps argument parsing predictable and avoids an implicit `/bin/sh -c` wrapper.

Consider the difference between the two forms below. The first is the **actual** line in your file. The second is a conceptual alternative; do not substitute it in this exercise.

ACTUAL: EXEC-FORM CMD

```
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

CONCEPTUAL ONLY: SHELL-FORM CMD

```
CMD uvicorn app.main:app --host 0.0.0.0 --port 8000
```

A shell-form command may place the shell in PID 1, which complicates signal forwarding. The exec form here lets the application server receive the stop signal directly. Also notice `--host 0.0.0.0`: this binds Uvicorn to all **interfaces inside the container**. It does not determine which interfaces on your laptop are exposed—that is controlled separately by Docker port publishing.

6.8 Publish and Smoke-Test the Minimal API

Create a running container from your image. Binding the host side to `127.0.0.1` confines this exercise to the local machine on supported setups; the container still listens on 8000. Name the container to make inspection and cleanup explicit.

START THE CONTAINER (ONE LINE)

```
docker run -d --name db-starter -p 127.0.0.1:8000:8000 deployboard-starter:0.1.0
```

VERIFY STATUS AND LOGS

```
docker ps --filter name=db-starter
docker logs db-starter
```

Docker will display the running container, its published-port mapping, and Uvicorn startup messages. Give the service a moment to start, then open `http://127.0.0.1:8000/health` and `http://127.0.0.1:8000/hello` in your browser. FastAPI also provides interactive API documentation at `http://127.0.0.1:8000/docs`; this is a learning convenience, not an authenticated production interface.

MACOS / LINUX: TEST HTTP ENDPOINTS

```
curl -fsS http://127.0.0.1:8000/health
curl -fsS http://127.0.0.1:8000/hello
```

WINDOWS POWERSHELL: TEST HTTP ENDPOINTS

```
Invoke-RestMethod http://127.0.0.1:8000/health
Invoke-RestMethod http://127.0.0.1:8000/hello
```

EXPECTED OUTPUT / EXPECTED RESPONSE BODIES (JSON)

```
{"status":"ok","service":"deployboard-starter"}
{"message":"Hello from DeployBoard","stage":"local"}
```

PowerShell may display JSON as formatted properties instead of a single line, and JSON field spacing or order can differ. Verify values, not typography. Try `/missing`: the app should respond with HTTP 404. **Do not test on a shared network host without understanding port exposure and local firewall rules.**

> **IF THE PAGE DOES NOT LOAD** Check `docker ps -a --filter name=db-starter`, then `docker logs db-starter`. If the container is absent, check your image tag. If it exits, read the traceback. If the container is healthy but the port is unavailable, check whether another process already uses port 8000 and whether your host URL includes `127.0.0.1`.

6.9 Understand Signals and Application Shutdown

A container is not a miniature desktop. Its lifetime follows its main process. When Docker stops the container, it sends the configured stop signal (normally `SIGTERM`) and allows a grace period before forcing termination. Because our command directly starts Uvicorn, the server can handle the signal and finish shutting down.

OBSERVE THE PROCESS, THEN STOP

```
docker top db-starter
docker stop --time 10 db-starter
docker ps -a --filter name=db-starter
docker logs --tail 12 db-starter
```

The container should move to an exited state, and its logs should normally include Uvicorn shutdown messages. Exit code and timing can vary, so do not teach a single exit number as a universal signal-handling guarantee. Inspect the actual result instead.

CHECK STOPPED STATE

```
docker inspect --format "{{.State.Status}}" db-starter
docker inspect --format "{{.State.ExitCode}}" db-starter
```

Then start the **same container**, rather than creating a replacement:

RESTART THE EXISTING CONTAINER

```
docker start db-starter
```

```
docker ps --filter name=db-starter
```

The host port mapping and container configuration remain attached to the stopped container. Later, when code changes, you will normally build a new image and replace a container. For this lifecycle experiment, reusing the existing container makes the distinction visible.

6.10 Use Build-Time and Runtime Configuration Correctly

Not every configurable value belongs in the same place. `ARG BUILD_VERSION` belongs to the **build** and becomes a label in our Dockerfile. `APP_GREETING` and `APP_STAGE` are **runtime** settings read by the Python service when it handles `/hello`. Changing runtime values must not require another image build.

Setting	When supplied	Where it is observed
BUILD_VERSION=0.1.1	docker build --build-arg	Image label: org.opencontainers.image.version
APP_GREETING=Welcome...	docker run -e	Response from GET /hello
APP_STAGE=demo	docker run -e	Response from GET /hello

BUILD A METADATA-VERSIONED IMAGE

```
docker build --build-arg BUILD_VERSION=0.1.1 -t deployboard-starter:0.1.1 .
docker image inspect --format '{{json .Config.Labels}}' deployboard-starter:0.1.1
```

The labels output includes `org.opencontainers.image.version` with the value `0.1.1`. The single-quoted Go template is designed to work in Bash and PowerShell. The tag and the label are distinct: the tag selects an image reference; the label records metadata.

RUN A SECOND INSTANCE WITH OVERRIDES (ONE LINE)

```
docker run -d --name db-starter-demo -p 127.0.0.1:8001:8000 -e APP_GREETING="Welcome to DeployBoard" -e APP_STAGE=demo deployboard-starter:0.1.1
```

MACOS / LINUX: CHECK THE OVERRIDE

```
curl -fsS http://127.0.0.1:8001/hello
```

EXPECTED OUTPUT / EXPECTED RESPONSE BODY

```
{"message":"Welcome to DeployBoard","stage":"demo"}
```

For Windows PowerShell, use `Invoke-RestMethod http://127.0.0.1:8001/hello`. You now have two containers from closely related images, with different runtime configurations and different host ports. Neither writes persistent data, and neither requires a database.

> **SECRETS ARE DIFFERENT** Environment variables are easy to inspect through container metadata and can leak through diagnostics. Neither `ARG` nor ordinary `ENV` is a safe place for passwords or tokens. Chapter 19 introduces build secrets and runtime secret handling.

6.11 Checkpoint: Ship the Stateless DeployBoard Starter Image

Treat the checkpoint as a small handoff to another engineer. They should be able to enter your directory, build the image from the printed files, start it without installing FastAPI on the host, and prove its behavior. Complete the following checks before leaving the chapter.

Acceptance evidence

Evidence	Proof command or observation
Source complete	Five files match the listings: Dockerfile, .dockerignore, requirements.txt, app/__init__.py, app/main.py.
Image built	docker image ls deployboard-starter
Non-root identity	docker image inspect --format "{{.Config.User}}" deployboard-starter:0.1.0
Contract correct	GET /health and GET /hello return expected JSON
Port safely bound	docker ps shows 127.0.0.1:8000->8000/tcp
Runtime settings work	GET /hello on 8001 returns demo greeting and stage
Stop behavior observed	docker stop --time 10 db-starter; logs and exited state inspected
Cleanup isolated	Only named db-starter containers removed; no global prune

When a check fails

Symptom	First reliable action
COPY fails during build	Confirm terminal is in project root; check case-sensitive file names.
No module named app	Confirm app/main.py exists, WORKDIR is /srv/deployboard, COPY path is app/.
Port 8000 already allocated	Stop the conflicting service or map host port 8002:8000 instead.
Forbidden / permission issue	Inspect USER, file ownership, and whether your code expects writable directories.
Changes do not appear	Rebuild the image and create a new container; restarting an old one does not update its image.

Clean up only this chapter's resources

REMOVE THE TWO NAMED CONTAINERS

```
docker rm -f db-starter db-starter-demo
```

The two local images may be retained for later comparison. If you want to delete only these images after verifying the containers are gone, run the following optional command. Do not use `docker system prune --volumes` as a substitute for named cleanup.

OPTIONAL: DELETE ONLY THESE CHAPTER IMAGES

```
docker image rm deployboard-starter:0.1.0 deployboard-starter:0.1.1
```

You have crossed the important boundary from a static-site image to a running application image. The source is copied into the artifact; Python dependencies are installed during the build; the process runs under a dedicated account; and HTTP responses can be changed at runtime. In Chapter 7, you will measure the cost of this build and deliberately improve it.

Chapter 7 — Build Faster and Smaller with BuildKit

A Docker image can work perfectly and still be expensive to rebuild. In Chapter 6, you packaged a small FastAPI service. Now imagine making a one-line change to its greeting, only to watch the builder reinstall packages that never changed. This workshop shows you how to distinguish necessary work from avoidable work, then prove that your changes improve the build.

WHAT YOU WILL SHIP Three runnable image variants, a two-stage production Dockerfile, a non-root running API, and a measurement sheet that records what actually improved. No registry account, cloud host, or external repository is required.

LAB IDENTITY Work only in a new `deployboard-buildkit` directory. All containers, tags, and a dedicated Buildx builder use the `db-ch7` prefix to avoid touching Chapters 1–6 or unrelated Docker projects.

7.1 Understand BuildKit, Builders, and Buildx

BuildKit is Docker's modern build engine: it translates Dockerfile instructions and their inputs into work that can be cached or repeated. **Buildx** is the CLI interface for choosing and using builders. A **builder** is the BuildKit instance that performs the work; the ordinary `docker` driver uses the engine-integrated backend, while the `docker-container` driver runs its own BuildKit container. These are related components, not interchangeable names.

VERIFY THE INSTALLED TOOLS

```
docker version
docker buildx version
docker buildx ls
```

Find the builder currently marked with `*`. We will create an isolated builder for this chapter so that any deliberate cache deletion remains scoped to the lab. The following two commands create it and start its BuildKit service; if you already created a builder named `db-ch7-lab` during this chapter, skip the first command.

CREATE A DEDICATED LAB BUILDER

```
docker buildx create --name db-ch7-lab --driver docker-container
docker buildx inspect db-ch7-lab --bootstrap
```

A `docker-container` builder does **not** automatically make its image available to `docker run`. Every runnable build in this chapter therefore includes `--load`. We specify `--builder db-ch7-lab` rather than changing your global builder selection. If your environment cannot create another builder, you can use your existing local Buildx builder and omit the `--builder` option—but do not run the lab's scoped prune command against a shared builder.

EXPECTED EVIDENCE The inspect command reports a functioning BuildKit instance and one or more Linux platforms. The first invocation may download the BuildKit helper image. That initial download is setup time, not an application build-time measurement.

7.2 Inspect the Build Context and Layer Cache

Start with the exact stateless API from Chapter 6. The **build context** is the directory whose files the builder is allowed to access after ignore rules are applied. It is not the same thing as the Dockerfile location. A smaller context reduces unnecessary file transfer and prevents local clutter from affecting the cache.

CREATE THE PROJECT DIRECTORY (MACOS / LINUX)

```
mkdir -p deployboard-buildkit/app
cd deployboard-buildkit
```

CREATE THE PROJECT DIRECTORY (WINDOWS POWERSHELL)

```
New-Item -ItemType Directory -Force deployboard-buildkit/app
Set-Location deployboard-buildkit
```

Create these files exactly. The app retains the Chapter 6 `/health` and `/hello` contract; no databases, authentication, or queues are introduced here.

FILE: APP/__INIT__.PY

```
"""DeployBoard starter application package."""
```

FILE: APP/MAIN.PY

```
"""Stateless DeployBoard starter carried forward from Chapter 6."""

import os
from fastapi import FastAPI

app = FastAPI(title="DeployBoard Starter", version="0.1.0")

@app.get("/health")
def health() -> dict[str, str]:
    return {"status": "ok", "service": "deployboard-starter"}

@app.get("/hello")
def hello() -> dict[str, str]:
```

```
    return {
        "message": os.getenv("APP_GREETING", "Hello from DeployBoard"),
        "stage": os.getenv("APP_STAGE", "local"),
    }
```

FILE: REQUIREMENTS.TXT

```
fastapi==0.146.0
uvicorn==0.53.0
```

FILE: .DOCKERIGNORE

```
.git
.venv
__pycache__
*.pyc
.env
*.pem
*.key
.pytest_cache
.DS_Store
Dockerfile.*
*.log
```

The `Dockerfile.*` exclusion keeps experimental Dockerfiles out of `COPY . .`, but Docker still sends the explicitly selected Dockerfile to its build frontend. Do not put tokens, SSH keys, or other secrets in the context: ignore rules reduce accidental inclusion but are not a substitute for proper secret handling.

EXAMINE STORAGE AND AVAILABLE BUILD COMMANDS

```
docker buildx du --builder db-ch7-lab
docker buildx build --help
```

At the beginning, disk use may be low or zero. Later, `docker buildx du` reveals cache growth. Unlike `docker image ls`, it describes builder cache rather than only loaded runtime images.

7.3 Arrange Dockerfile Instructions for Effective Caching

Compare two versions. The first is intentionally inefficient: it copies the complete application before installing dependencies. The second preserves the good ordering introduced in Chapter 6 by copying only `requirements.txt` before the installation step. Both are runnable, non-root containers, so the experiment changes the build strategy rather than the API behavior.

FILE: DOCKERFILE.NAIVE (INTENTIONALLY INEFFICIENT)

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim-bookworm
WORKDIR /srv/deployboard
COPY . .
RUN python -m pip install --disable-pip-version-check -r requirements.txt
RUN groupadd --gid 10001 appgroup \
   && useradd --uid 10001 --gid appgroup \
      --create-home --shell /usr/sbin/nologin appuser
USER appuser
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

FILE: DOCKERFILE.CACHED (CHAPTER 6 BASELINE)

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim-bookworm
ENV PYTHONDONTWRITEBYTECODE=1 \
   PYTHONUNBUFFERED=1
WORKDIR /srv/deployboard
COPY requirements.txt ./
RUN python -m pip install --no-cache-dir --disable-pip-version-check \
   -r requirements.txt
RUN groupadd --gid 10001 appgroup \
   && useradd --uid 10001 --gid appgroup \
      --create-home --shell /usr/sbin/nologin appuser
COPY --chown=appuser:appgroup app/ ./app/
USER appuser
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

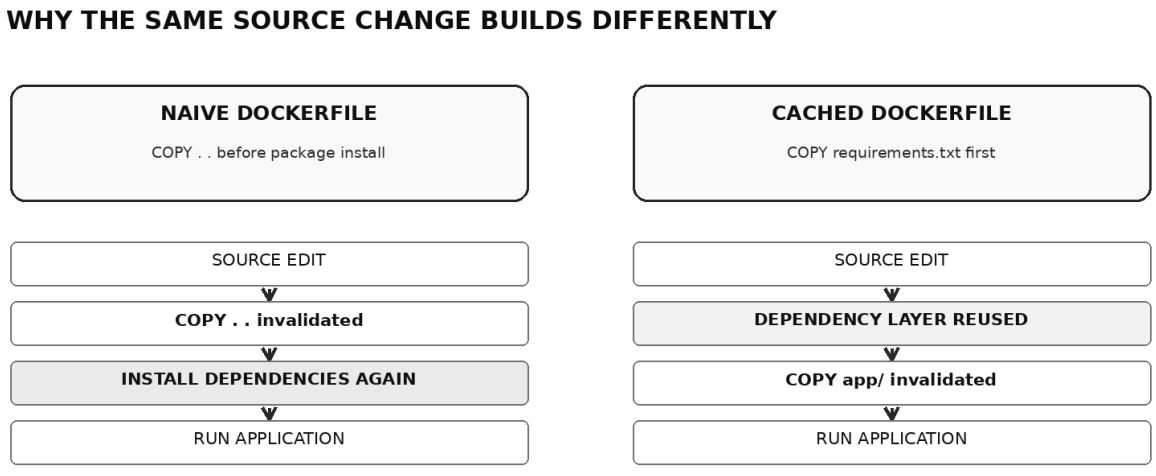


Figure 7.1 — A source edit invalidates more work when the copy instruction precedes dependency installation.

Build both images once. Use a stopwatch to note the **end-to-end elapsed time**, including transfers and image loading. The first build may spend time

downloading the base image and Python packages; that is not a valid cached-build comparison.

BUILD BOTH BASELINES

```
docker buildx build --builder db-ch7-lab --load --progress=plain -f Dockerfile.naive -t
deployboard:naive .
docker buildx build --builder db-ch7-lab --load --progress=plain -f Dockerfile.cached -t
deployboard:cached .
```

Repeat both commands unchanged. In the plain logs, look for `CACHED` on completed instructions. Now change only the default greeting in `app/main.py` from `Hello from DeployBoard` to `Hello from DeployBoard BuildKit`, and rebuild both. The naive file must repeat its dependency-installation instruction; the cached variant should reuse that earlier layer because `requirements.txt` did not change. After recording the result, restore the original greeting so the final application source matches Chapter 6 again.

CRITICAL DISTINCTION The cache is not a stopwatch and it is not a freshness checker. A cached dependency install does not mean pip rechecked a repository for newer package files. Dependency updates require an intentional input change, base-image update, or controlled rebuild.

7.4 Use Cache Mounts for Package Installation

A layer cache asks whether the entire command can be reused. A **cache mount** holds reusable download data even when the command must run again. Python pip's download cache is a good example: if the install step needs to repeat, it can reuse previously fetched packages rather than downloading everything again.

BUILDKIT CACHE-MOUNT PATTERN

```
RUN --mount=type=cache,target=/root/.cache/pip \
  /opt/venv/bin/python -m pip install \
  --disable-pip-version-check -r requirements.txt
```

This instruction belongs in the builder stage shown next. The mount at `/root/.cache/pip` is temporary to the build step and stored in the builder cache, not copied into the final runtime filesystem. Do not combine this pattern with `--no-cache-dir`; that option tells pip not to use its download cache and defeats the point of the mount.

Cache mounts are accelerators, not required build inputs. Builds must continue working when the cache is empty or has been evicted. For deterministic releases, pin dependency versions, and where appropriate use verified dependency hashes or a fully locked requirements file. Merely

pinning two top-level packages does not guarantee that every transitive dependency version is frozen.

7.5 Introduce Multi-Stage Builds and Named Stages

The optimized Dockerfile separates two responsibilities. A stage named `builder` creates a virtual environment and installs Python dependencies; a stage named `runtime` contains the Python runtime, the installed virtual environment, your app, and the non-root service account. A named stage can be referenced explicitly using `COPY --from=builder`, which is clearer than referring to numbered stages.

FILE: DOCKERFILE (THE FINAL OPTIMIZED IMAGE)

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim-bookworm AS builder
WORKDIR /build
RUN python -m venv /opt/venv
COPY requirements.txt ./
RUN --mount=type=cache,target=/root/.cache/pip \
   /opt/venv/bin/python -m pip install \
   --disable-pip-version-check -r requirements.txt

FROM python:3.13-slim-bookworm AS runtime
ENV PYTHONDONTWRITEBYTECODE=1 \
   PYTHONUNBUFFERED=1 \
   PATH="/opt/venv/bin:$PATH"
WORKDIR /srv/deployboard
COPY --from=builder /opt/venv /opt/venv
RUN groupadd --gid 10001 appgroup \
   && useradd --uid 10001 --gid appgroup \
      --create-home --shell /usr/sbin/nologin appuser
COPY --chown=appuser:appgroup app/ ./app/
ARG BUILD_VERSION=0.2.0
LABEL org.opencontainers.image.version="${BUILD_VERSION}"
USER appuser
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

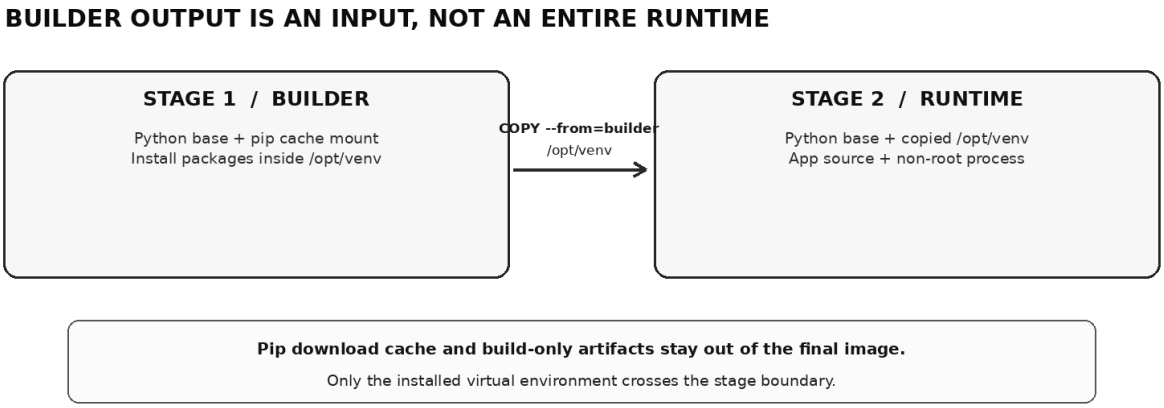


Figure 7.2 — The final stage receives the installed environment but not the builder's download cache.

Using the same Python base family in both stages avoids copying a virtual environment between incompatible runtime distributions or Python versions. The explicit `PATH` makes the installed `uvicorn` command available without invoking a host Python interpreter. This still uses ordinary Docker image layers: a multi-stage build is not a new container runtime.

BUILD AND LOAD THE OPTIMIZED IMAGE

```
docker buildx build --builder db-ch7-lab --load --progress=plain -f Dockerfile -t
deployboard:optimized .
```

Look for stage names such as `builder` and `runtime` in the build log. The final tagged image is the runtime stage. The builder stage's package downloads are not embedded in that exported runtime image.

7.6 Separate Builder Tools from the Runtime Image

The final container does not need the builder's pip-download cache, an active build daemon, or the source-code tooling used only during construction. `COPY --from=builder /opt/venv /opt/venv` transfers only the installed environment required to run the service. This is a narrower transfer boundary than copying the entire `/build` directory.

For this lightweight FastAPI example, no compiler is needed, and the final optimized image is **not guaranteed** to be smaller than the already-conscious Chapter 6 baseline. Multi-stage design offers a clean separation even when the size difference is minor. More dramatic savings typically

appear when native build tools, temporary test dependencies, or compiled artifacts are excluded from the runtime.

CHECK PROCESS IDENTITY AND ENDPOINT

```
docker run -d --name db-ch7-app -p 127.0.0.1:8767:8000 deployboard:optimized
docker exec db-ch7-app id -u
docker exec db-ch7-app python -c "import urllib.request;
print(urllib.request.urlopen('http://127.0.0.1:8000/health').read().decode())"
```

Expected results: the UID is `10001`; the health response is `{"status":"ok","service":"deployboard-starter"}` (JSON spacing may vary). Open `http://127.0.0.1:8767/hello` in a browser and confirm the greeting. The port is bound to host loopback, not deliberately published on every network interface.

STOP AND REMOVE ONLY THE LAB CONTAINER

```
docker stop db-ch7-app
docker rm db-ch7-app
```

7.7 Compare Image Size, Build Time, and Dependencies

"Faster" and "smaller" need separate evidence. Measure the three images and compare repeated builds under roughly identical conditions. Cache warmth, network state, machine architecture, and previous downloads influence measurements. Never print a fabricated benchmark number as if it came from your machine.

GET IMAGE SIZES AND LAYER HISTORIES

```
docker image ls deployboard
docker image inspect --format "{{.Size}}" deployboard:naive
docker image inspect --format "{{.Size}}" deployboard:cached
docker image inspect --format "{{.Size}}" deployboard:optimized
docker image history deployboard:optimized
```

Variant	Fresh build (s)	Repeat build (s)	After source edit (s)	Image size (MB)
Naive	____	____	____	____
Cached	____	____	____	____
Optimized	____	____	____	____

For each image, divide `.Size` in bytes by 1,048,576 to obtain MiB. If reporting decimal MB instead, divide by 1,000,000 and label it correctly. Inspect the plain build logs alongside the timings; a small apparent speedup may simply be measurement noise. If the optimized image is larger, report it and explain that the extra virtual-environment layout can outweigh savings for a small dependency set.

MAKE A FAIR COMPARISON Run the builds on one Docker context, with the same platform, package pins, builder, source tree, and approximate network conditions. Do not compare a first-ever image pull with a warm rebuild and call it a Dockerfile optimization.

7.8 Use Dockerfile Build Checks and Lint Results

BuildKit can examine your Dockerfile without executing the full build. Run a check before accepting the optimized variant. The exact rule set depends on the installed Dockerfile frontend and Buildx versions; a clean check is useful evidence, not a security audit.

CHECK THE FINAL DOCKERFILE

```
docker buildx build --builder db-ch7-lab --check -f Dockerfile .
```

For practice, inspect an intentionally questionable file. The uppercase stage name and shell-form `CMD` can trigger checks such as `StageNameCasing` and `JSONArgsRecommended` in supported versions. If a warning is absent, consult the installed frontend's check list rather than assuming the feature is broken.

FILE: DOCKERFILE.BADCHECK (DIAGNOSTIC ONLY)

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim-bookworm AS BUILDER
CMD echo hello
```

RUN A DELIBERATE CHECK

```
docker buildx build --builder db-ch7-lab --check -f Dockerfile.badcheck .
```

A check invocation with violations can exit nonzero. This file is **not** used to build or run an application. Fix its warnings conceptually—lowercase stage name and an exec-form command—then return to the production `Dockerfile`.

7.9 Avoid Invalid Cache Assumptions and Hidden Dependencies

Caching introduces two failure modes: incorrect expectations about invalidation and hidden dependencies on previously built artifacts. Docker decides whether a step is reusable from the instruction and the inputs that

BuildKit tracks. The age of a package repository does not automatically invalidate a cached `RUN pip install` instruction; changing an untracked website or API response does not magically rerun it.

Misleading assumption	What to do instead
A cached pip install checks for new dependencies.	Update pins or a lockfile explicitly; rebuild and test.
Changing any file must reinstall packages.	COPY narrow dependency inputs before frequently edited source.
A cache mount is part of the final image.	Verify runtime layers; treat builder cache as disposable.
Every multi-stage image is smaller.	Measure actual exported images and inspect runtime contents.
Build ARG or ENV can safely store credentials.	Use BuildKit secret mounts for build-time secrets in later security labs.

A `.dockerignore` pattern also needs care: excluding your application, `requirements.txt`, or required test fixtures will make builds fail or silently omit files. Prefer explicitly copying known inputs and verify them with an uncached build. Later chapters will cover digests, locked dependencies, attestations, and supply chain checks in greater depth.

7.10 Rebuild from a Clean Cache to Prove Reliability

A successful warm build proves that BuildKit can reuse known work. It does **not** prove that another machine can build the application. Use the builder created exclusively for this chapter to remove its cache, then rebuild the image. Do not run an indiscriminate `docker system prune`, `docker image prune -a`, or a cache prune against a shared builder: those commands could remove unrelated work.

INSPECT AND PRUNE ONLY THE LAB BUILDER CACHE

```
docker buildx du --builder db-ch7-lab
docker buildx prune --builder db-ch7-lab --all --force
docker buildx du --builder db-ch7-lab
```

COLD REBUILD AND RUNTIME SMOKE TEST

```
docker buildx build --builder db-ch7-lab --load --progress=plain --no-cache -f Dockerfile -t
deployboard:optimized-clean .
docker run -d --name db-ch7-clean -p 127.0.0.1:8767:8000 deployboard:optimized-clean
docker exec db-ch7-clean python -c "import urllib.request;
print(urllib.request.urlopen('http://127.0.0.1:8000/health').read().decode())"
```

```
docker stop db-ch7-clean
docker rm db-ch7-clean
```

Why use both prune and `--no-cache`? The prune removes existing **lab-builder cache state**, including cache mounts; the flag makes it explicit that the new build must not reuse earlier instruction results. `--no-cache` alone does not promise empty cache mounts. This drill proves clean-cache buildability, but true cross-machine reproducibility also requires a locked dependency graph, pinned base-image digest, and environment/version documentation (Chapter 8).

IF IT FAILS Check the exact dependency installation error first. A fresh build may require network access to the registry and Python package index. Confirm disk space, proxy settings, and image architecture. Do not "fix" the problem by silently changing dependency versions.

7.11 Checkpoint: Optimize DeployBoard with Measurements

You are now the release engineer for the starter API. A teammate wants a faster rebuild but will reject a change that breaks startup, changes the endpoints, runs as root, or requires previously cached artifacts. Your job is to submit evidence, not assurances.

Acceptance evidence

1. **Function:** Both endpoints respond with the Chapter 6 contract, and the container runs as UID 10001. 2. **Build:** `Dockerfile` has two named stages, a cache-mounted pip installation, and a narrow stage-to-stage copy. 3. **Diagnostics:** The BuildKit check runs, and you can explain any warning. 4. **Measurements:** Your worksheet contains comparable elapsed times and measured image sizes. 5. **Rebuild:** A cold build and endpoint smoke test pass without relying on prior lab cache.

Release decision

Explain, in three sentences, which variant you would ship, whether it is actually smaller on your machine, and which risk still remains. A defensible decision may favor the multi-stage image for isolation even when its size is marginally greater; an engineering report should state the tradeoff honestly.

If the benchmark disagrees with your prediction

A fast second build does not automatically validate good Dockerfile ordering: perhaps both variants were already cached. Rebuild after modifying only `app/main.py`, then read which specific instruction became `CACHED` and which reran. If the naive dependency layer is unexpectedly cached, make sure `COPY . .` actually includes the modified file and that you are building from the intended project directory. If the optimized image is unexpectedly large, inspect its exported layers and installed files rather than assuming the builder stage was exported.

Symptom	Evidence to collect	First corrective action
Tagged image missing	Build output shows only cache result	Rebuild with `--load` for docker-container driver
Warm build reinstalls packages	Plain logs show dependency RUN rerun	Confirm requirements are copied before app/
Port 8767 unavailable	docker ps / port error	Stop your own conflicting container or choose a new loopback port
Optimized build fails cold	Fresh pip/base-image fetch error	Check pinned inputs, registry access, platform, available space

Release evidence sheet

Proof item	What you observed
Source revision tested	_____
Chosen image tag / base tag	_____
Measured warm-build improvement	_____
Measured image-size difference	_____
Non-root UID and endpoint checks	_____
Clean-cache rebuild outcome	_____

Keep the completed sheet with your Dockerfiles. The numbers belong to your particular builder, network, and machine; do not transfer them into another environment as universal promises.

OPTIONAL CLEANUP: ONLY THIS CHAPTER'S OBJECTS

```
docker image rm deployboard:naive deployboard:cached deployboard:optimized
deployboard:optimized-clean
docker buildx rm db-ch7-lab
```

Run this only after you have finished measuring and verified that none of these named images or builders is needed elsewhere. `docker buildx rm` removes the dedicated lab builder and its stored state; it does not delete

unrelated builders. If an image is still used by a container, Docker can refuse removal; inspect that container before taking further action.

NEXT DISPATCH Chapter 8 will make builds identifiable and portable through tags, digests, platform inspection, and explicit release metadata. Do not mistake a fast local rebuild for a reproducible release: this chapter established the baseline for proving one.

Chapter 8 — Make Images Portable, Traceable, and Reproducible

Your optimized DeployBoard image now builds and starts. But imagine receiving a deployment report that says only “running version 0.3.0.” Which bytes are running? Was the base image updated? Does the image also support an ARM server? Can someone reconstruct the build a month later? This chapter teaches you to answer those questions with verifiable evidence—not assumptions.

Command convention: multiline commands below use Bash backslash continuation (valid in macOS Terminal, Linux shells, and WSL). In PowerShell, join those lines into one command without the trailing backslashes; do not paste Bash continuations directly into PowerShell.

The goal is one identifiable release of the stateless DeployBoard starter from Chapters 6 and 7. You will keep the application small so the release mechanics remain visible. PostgreSQL, Redis, and production rollout come later; here, the artifact itself is the product.

> **YOUR DELIVERABLE** A locally runnable tagged image; a recorded base-image digest and resolved dependency inventory; a multi-platform OCI artifact if your builder supports both targets; and a release-evidence record. Publishing to a registry is optional.

8.1 Choose Tags, Immutable References, and Digests

An image repository groups related images, while a tag is a human-friendly pointer to a release. A tag such as deployboard-starter:0.3.0 is useful for people, but can be moved. A digest is a content-addressed identifier beginning with sha256:. Once an exact manifest is selected by digest, the reference names specific content. Do not confuse the local image ID, a platform-specific manifest digest, and the digest of a multi-platform index.

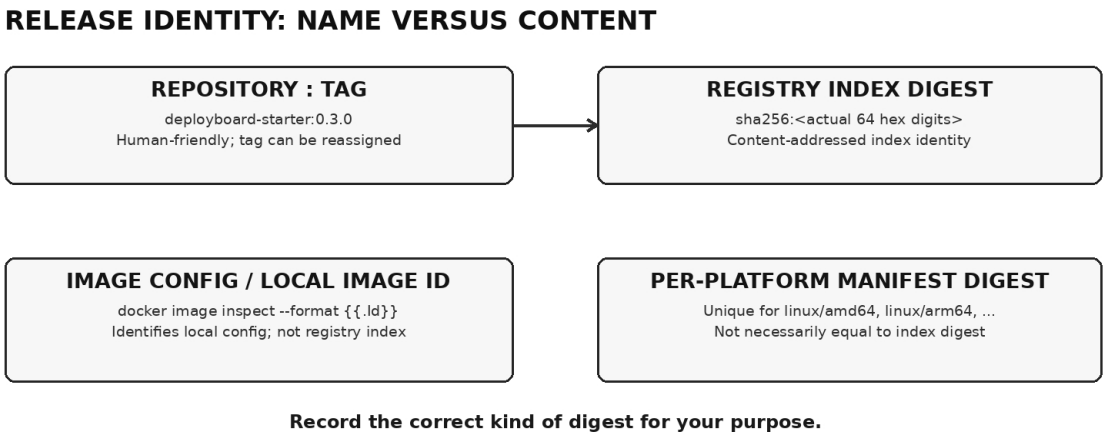


Figure 8.1 — Repository tags, image IDs, image manifests, and index digests serve different purposes.

INSPECT AN EXISTING CHAPTER 7 BUILD

```
docker image ls deployboard-starter
docker image inspect deployboard-starter:0.2.0 --format "{{.Id}}"
```

The second command succeeds if you retained the image tagged 0.2.0 in Chapter 7. Otherwise, build the release later in Section 8.8 and substitute that tag. The long sha256: value returned by .Id is the local image configuration digest; it is not proof of a published registry index.

OPTIONAL: DEMONSTRATE TAG ALIASING

```
docker image tag deployboard-starter:0.2.0 deployboard-starter:lab-copy
docker image inspect deployboard-starter:lab-copy --format "{{.Id}}"
```

INTERPRETATION Both tags initially point to the same image ID; neither tag is intrinsically immutable. Do not use tag equality as evidence that two remote deployments contain identical bytes.

8.2 Pin Base Images and Plan Safe Updates

The Chapter 7 Dockerfile used python:3.13-slim-bookworm. This is a specific major/minor Python and Debian-family image label, but it is still a mutable tag. For a controlled release, select a reviewed index digest and use the exact reference in both builder and runtime stages. A digest pin improves repeatability; it does not guarantee that the pinned contents are vulnerability-free.

READ THE CURRENT REGISTRY INDEX

```
docker buildx imagetools inspect python:3.13-slim-bookworm
```

Record the Digest field at the top of the response. On platforms where this tag is an image index, the listed linux/amd64 and linux/arm64 entries may each have a different child digest. Use the top-level index digest for a shared multi-platform base reference. The actual digest changes over time, so this book does not invent one.

REFERENCE FORMAT — REPLACE THE DIGEST BEFORE RUNNING

```
docker buildx build --builder db-release-builder --platform linux/amd64 \
  --build-arg BASE_IMAGE=python:3.13-slim-
bookworm@sha256:REPLACE_WITH_INDEX_DIGEST \
  -t deployboard-starter:0.3.0 --load .
```

The example shows the syntax of a pinned build. Copy the genuine 64-character digest from the registry inspection and replace REPLACE_WITH_INDEX_DIGEST; do not type that illustrative text literally. The builder is created in Section 8.5. For your first pass, use the default BASE_IMAGE and capture the digest in the evidence sheet; repeat with the verified digest after the builder setup.

> **MAINTENANCE DECISION** Store the old pinned digest with your release record. When a new base is published, review its changes, rebuild deliberately, run smoke tests, and only then move the project to the new digest. This balances repeatability with security updates.

8.3 Inspect Layers, History, Labels, and Manifests

A release review should distinguish the configuration stored in the image, the filesystem layers it refers to, and the registry manifest that distributes it. The following commands are observational: they will not change your image.

INSPECT THE LOCAL RELEASE AFTER SECTION 8.8

```
docker image history --no-trunc deployboard-starter:0.3.0
docker image inspect deployboard-starter:0.3.0 \
  --format "{{json .Config.Labels}}"
docker image inspect deployboard-starter:0.3.0 \
  --format "{{.Os}}/{{.Architecture}}"
```

History explains the lineage of image instructions; it does not expose every runtime behavior or provide a security audit. The labels are metadata

supplied during the build, not cryptographic authentication. For a remote image index, inspect the registry directly:

REGISTRY INSPECTION EXAMPLE

```
docker buildx imagetools inspect python:3.13-slim-bookworm
```

Do not expect imagetools inspect to discover a local-only image tag. The command fetches a registry image, which is why the published Python base is used here. Your DeployBoard registry inspection becomes possible after you publish it in Chapter 25 or by taking the optional registry path in Section 8.6.

8.4 Distinguish CPU Architecture and Operating System

A container image includes an operating-system and architecture expectation. The common Linux targets are linux/amd64 (x86-64 servers) and linux/arm64 (64-bit ARM devices). A Windows or macOS computer running Docker Desktop typically hosts Linux containers inside a Linux environment; that does not turn a Linux image into a native Windows or macOS container.

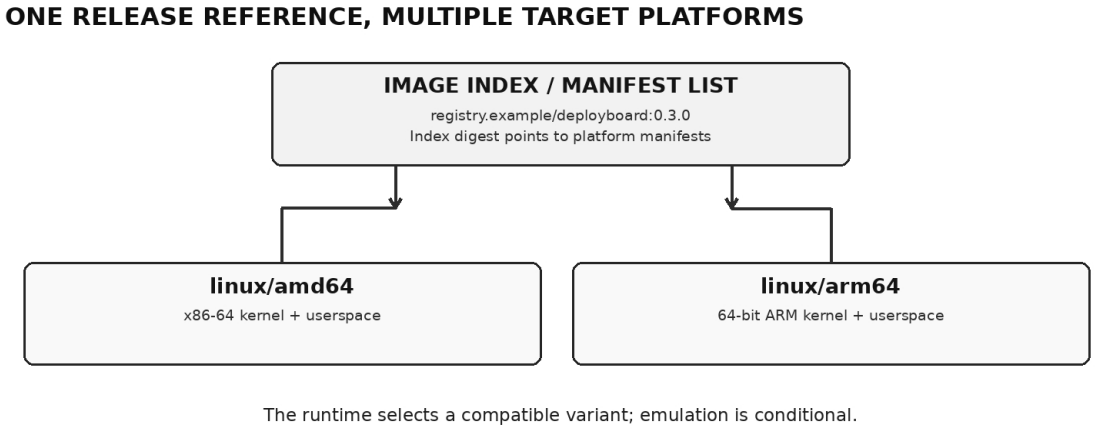


Figure 8.2 — A registry image index can point to distinct Linux variants for AMD64 and ARM64.

COMPARE THE DAEMON AND IMAGE

```
docker version --format "Server: {{.Server.Os}}/{{.Server.Arch}}"
docker image inspect deployboard-starter:0.3.0 \
  --format "Image: {{.Os}}/{{.Architecture}}"
```

An ARM64 host may run an AMD64 image through supported emulation, but that is not native execution. If you only test on a single architecture, document that limitation. The two targets taught here are both Linux; building a Windows container requires a compatible Windows container base and a different runtime situation.

8.5 Create a Buildx Builder for Multi-Platform Builds

Buildx coordinates BuildKit builders. A dedicated docker-container builder is useful because it supports image-index exporters even on machines that still use the classic Docker image store. The new builder is isolated from the daemon image store, so builds are not automatically visible to docker image ls.

CREATE AND INSPECT YOUR RELEASE BUILDER

```
docker buildx version
docker buildx ls
docker buildx create --name db-release-builder \
  --driver docker-container --bootstrap --use
docker buildx inspect --bootstrap
```

Check the Platforms line. If linux/amd64 and linux/arm64 both appear, attempt the two-platform output in the next section. If one target is absent, use the single-platform fallback; platform support can depend on emulation availability, native nodes, and host configuration. Do not install privileged emulation helpers on a managed machine without authorization.

IDEMPOTENCE If db-release-builder already exists, do not create it again. Run docker buildx use db-release-builder followed by docker buildx inspect --bootstrap. To return to your previous builder, use docker buildx ls to identify it and docker buildx use its name.

8.6 Build and Inspect Multi-Platform Image Output

Use an OCI-layout tarball as the no-account path. The OCI exporter preserves the image-index structure without requiring a paid service or a public registry. It is different from docker save: a Docker image archive

produced by the daemon and an OCI layout tar produced by Buildx have different intended workflows.

BUILD BOTH PLATFORMS TO A LOCAL OCI ARTIFACT

```
docker buildx build --builder db-release-builder \
  --platform linux/amd64,linux/arm64 \
  --tag deployboard-starter:0.3.0 \
  --output type=oci,dest=deployboard-multi.oci.tar .
```

Expected result: a nonempty deployboard-multi.oci.tar file with an index.json and OCI blobs. On Windows PowerShell, inspect the archive using tar -tf deployboard-multi.oci.tar; the same tar command works on modern macOS and Linux. The emitted index may also include attestation descriptors with unknown/unknown platform metadata. Those are not extra runnable architectures.

INSPECT THE ARCHIVE, THEN LOAD ONE PLATFORM FOR RUNNING

```
tar -tf deployboard-multi.oci.tar
docker buildx build --builder db-release-builder \
  --platform linux/amd64 --tag deployboard-starter:0.3.0 \
  --load .
```

With a classic image store, --load is for a single-platform result. A containerd image store may support multiple platforms, but the one-platform command above works reliably for this lab. If the two-platform build cannot run, complete a single-platform OCI build by changing --platform to linux/amd64 or to the platform shown in docker version.

Optional: inspect a published DeployBoard index

If you own an OCI registry account and want to publish now, replace YOUR_NAMESPACE in the next example and authenticate with docker login. Publishing may incur account or registry usage costs. Never publish private credentials or secrets in a public image.

OPTIONAL REGISTRY BRANCH — REQUIRES YOUR OWN NAMESPACE

```
docker buildx build --builder db-release-builder \
  --platform linux/amd64,linux/arm64 \
  -t YOUR_NAMESPACE/deployboard-starter:0.3.0 --push .
docker buildx imagetools inspect \
  YOUR_NAMESPACE/deployboard-starter:0.3.0
```

Record the index digest from the registry report. An index digest can change when annotations, manifests, or attached metadata change; record

the actual value reported for the released artifact rather than assuming it must match a local image ID.

8.7 Explain Emulation, Native Builders, and Tradeoffs

There are three useful strategies for more than one CPU architecture. Emulation may be the easiest but is often slower; a builder with native x86 and ARM nodes is faster for heavy compilation but costs more to administer; and cross-compilation is attractive when a language and toolchain support it. DeployBoard is a small Python service, so the aim here is correctness and familiarity rather than a benchmark competition.

Strategy	When it helps	What to verify
QEMU emulation	Learning or occasional cross-architecture output	Build succeeds and program starts on the target
Multiple native nodes	Frequent builds or compiled heavy dependencies	Builder node access and platform coverage
Cross-compilation	Supported compiled runtimes and toolchains	Binary target and runtime library compatibility

Python wheels can contain architecture-specific native code. A dependency set that installs on AMD64 is not automatically guaranteed to be available on ARM64. That is why each platform must be built and ideally smoke-tested, even when a multi-platform index was successfully created.

HONEST PERFORMANCE REPORTING If you time builds, record CPU, memory, host architecture, builder type, cache state, and network conditions. Avoid claiming a universal performance multiplier.

8.8 Control Rebuild Inputs and Dependency Versions

Reproducibility has degrees. Pinned direct package versions, a fixed base image digest, and a recorded build command make a build more explainable. They do not by themselves freeze transitive Python packages, index behavior, wheel selection, timestamps, or every build-tool implementation. Aim for a reconstructable release record now; byte-for-byte reproducibility is a separate, stricter test.

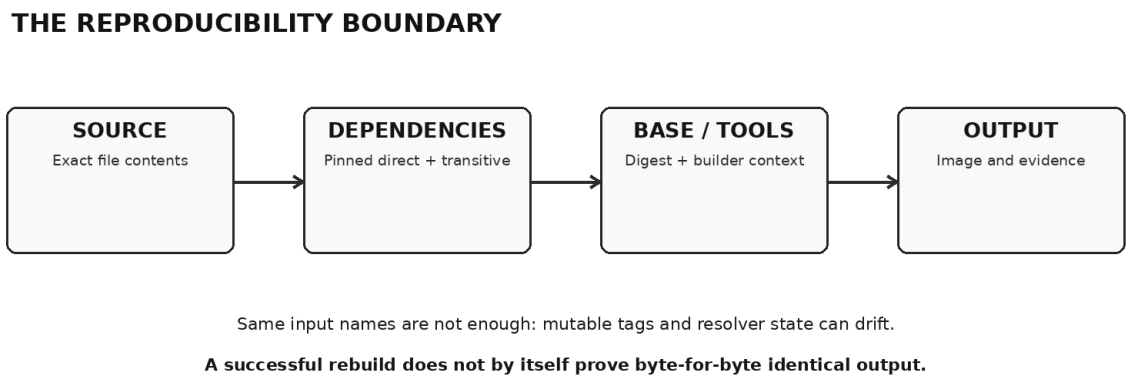


Figure 8.3 — Every mutable input weakens a claim that a later build is identical.

Create a project folder named deployboard-release with the exact five listings below. Keep the same stateless /health and /hello service from Chapter 7; the only material new features are the globally configurable base image and OCI release labels.

FILE: APP/MAIN.PY — FULL LISTING

```
"""Stateless DeployBoard API from Chapters 6 and 7."""
import os
from fastapi import FastAPI

app = FastAPI(title="DeployBoard Starter", version="0.1.0")

@app.get("/health")
def health() -> dict[str, str]:
    return {"status": "ok", "service": "deployboard-starter"}

@app.get("/hello")
def hello() -> dict[str, str]:
    return {
        "message": os.getenv("APP_GREETING", "Hello from DeployBoard"),
        "stage": os.getenv("APP_STAGE", "local"),
    }
```

FILE: APP/__INIT__.PY

```
"""DeployBoard starter application
package."""
```

FILE: REQUIREMENTS.TXT

```
fastapi==0.146.0
uvicorn==0.53.0
```

FILE: .DOCKERIGNORE

```
# Limit local build context and prevent credential leakage.
.git
.venv
__pycache__
*.pyc
.env
*.pem
*.key
.pytest_cache
.DS_Store
*.log
*.tar
*.oci.tar
release-evidence/
build-metadata.json
resolved-dependencies.txt
```

FILE: DOCKERFILE — COMPLETE RELEASE BUILD

```
# syntax=docker/dockerfile:1
# Override BASE_IMAGE with a VERIFIED index digest for frozen releases.
ARG BASE_IMAGE=python:3.13-slim-bookworm
FROM ${BASE_IMAGE} AS builder
WORKDIR /build
RUN python -m venv /opt/venv
COPY requirements.txt ./
RUN --mount=type=cache,target=/root/.cache/pip \
    /opt/venv/bin/python -m pip install \
    --disable-pip-version-check -r requirements.txt

FROM ${BASE_IMAGE} AS runtime
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PATH="/opt/venv/bin:$PATH"
WORKDIR /srv/deployboard
COPY --from=builder /opt/venv /opt/venv
RUN groupadd --gid 10001 appgroup \
    && useradd --uid 10001 --gid appgroup \
    --create-home --shell /usr/sbin/nologin appuser
COPY --chown=appuser:appgroup app/ ./app/
ARG RELEASE_VERSION=0.3.0
ARG SOURCE_REVISION=local-workshop
LABEL org.opencontainers.image.title="DeployBoard Starter" \
    org.opencontainers.image.description="Stateless release lab API" \
    org.opencontainers.image.version="${RELEASE_VERSION}" \
    org.opencontainers.image.revision="${SOURCE_REVISION}"
USER appuser
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

From the project root, build the tagged local image and supply truthful metadata. SOURCE_REVISION may be a repository commit when one exists, or local-workshop for the book-only path. A placeholder commit-looking value is worse than an honest local label.

BUILD AND START A PRIVATE LOCAL SMOKE TEST

```
docker buildx build --builder db-release-builder \
  --platform linux/amd64 --load \
  --build-arg RELEASE_VERSION=0.3.0 \
  --build-arg SOURCE_REVISION=local-workshop \
  -t deployboard-starter:0.3.0 .
docker run --rm -d --name db-ch08-api \
  -p 127.0.0.1:8088:8000 deployboard-starter:0.3.0
```

VERIFY THE APPLICATION WITHOUT DEPENDING ON HOST CURL

```
docker container ls --filter name=db-ch08-api
docker logs db-ch08-api
docker exec db-ch08-api python -c \
  "import urllib.request;
print(urllib.request.urlopen('http://127.0.0.1:8000/health').read().decode())"
```

The final command should print JSON containing "status":"ok" and "service":"deployboard-starter". Use docker rm -f db-ch08-api only after saving any needed logs; this stateless service stores no persistent user data. For a completely clean workspace, rerun the source files and image build rather than copying an existing container.

Capture resolved Python versions

INSPECT THE ACTUAL RESOLVED ENVIRONMENT

```
docker run --rm --entrypoint python \
  deployboard-starter:0.3.0 -m pip freeze
```

Copy the full output into your release notes as resolved-dependencies.txt. The requirements.txt file pins only the direct packages; pip freeze reveals the installed transitive versions on this platform. A proper repeatable dependency lock in a later production project should include transitive pins and hashes, regenerated when dependencies are updated; do not equate this evidence snapshot with a locked installer policy.

8.9 Add Useful OCI Labels and Build Metadata

OCI labels are a low-friction place to record a human-readable title, description, version, and source revision. In the supplied Dockerfile, labels

are attached to the final runtime stage, not only the builder stage. They can be inspected without starting the application.

INSPECT THE LABELS AND LOCAL IMAGE IDENTITY

```
docker image inspect deployboard-starter:0.3.0 \
  --format "{{json .Config.Labels}}"
docker image inspect deployboard-starter:0.3.0 \
  --format "{{.Id}}"
```

Expect to see org.opencontainers.image.title, .description, .version, and .revision keys. Build metadata is descriptive, not proof of origin or a signature. We will introduce attestations, vulnerability review, and signing tradeoffs in Chapter 20; do not claim a source revision label proves who built the image.

Store machine-readable build evidence

OPTIONAL BUILDX OUTPUT METADATA

```
docker buildx build --builder db-release-builder \
  --platform linux/amd64 --load \
  --metadata-file build-metadata.json \
  -t deployboard-starter:0.3.0 .
```

The build-metadata.json file contains builder-dependent fields that may include containerimage.digest and config digests. It is evidence about this build invocation; preserve it alongside the source snapshot and command log. Remember that .dockerignore keeps such extra local evidence out of the image context if you place it outside the project root or exclude its directory.

8.10 Export, Load, and Save Images When Appropriate

A saved Docker image archive is useful for moving a local image between Docker daemons. An OCI layout archive is a more format-oriented release artifact and may hold a multi-platform index. A registry is the normal option for distributing images across machines at scale. These formats and commands are not interchangeable merely because the files end in .tar.

DAEMON IMAGE ARCHIVE: SAVE AND LOAD

```
docker image save -o deployboard-amd64.docker.tar \
  deployboard-starter:0.3.0
docker image load -i deployboard-amd64.docker.tar
docker image inspect deployboard-starter:0.3.0 \
  --format "{{.Os}}/{{.Architecture}} {{.Id}}"
```

The saved archive contains the locally stored image variant(s), not a promise that it includes every architecture in a remote registry. On older engines, --platform options for save/load may be unavailable. Check docker image save --help and docker image load --help on the installed version.

Distribution method	Contains	Best use
docker image save / load	Local daemon image archive	Transport a local image to another Docker engine
Buildx OCI exporter	OCI layout, possibly multi-platform index	Inspect or transfer a standards-based build artifact
Registry push / pull	Manifest(s), layers, release references	Share named versioned releases across systems

ARCHIVE SECURITY A tar archive is not a safe substitute for a backup of volumes or databases. Images can contain software, metadata, and accidentally embedded credentials. Scan and control access to exported artifacts.

8.11 Checkpoint: Rebuild and Identify an Exact Release

The final challenge is to make a release someone else could evaluate without asking which unlabeled build you happened to run. Work from a clean copy of deployboard-release; do not depend on the running container from Chapter 7.

Release acceptance record

□ Run docker buildx inspect --bootstrap and record the builder and available target platforms.

□ Build deployboard-starter:0.3.0 for one supported Linux platform and record its local image ID.

□ Show the application returning a successful /health response while bound only to 127.0.0.1.

□ Inspect the four OCI labels; ensure the release version and source revision are truthful.

□ Record the real base-image index digest and Python dependency-resolution output.

□ Export a Docker archive and load it; confirm that the expected platform and image remain accessible.

□ If supported, export a multi-platform OCI archive and inspect its index contents.

□ Rebuild without assuming a warm cache; compare results and explain any differences.

RELEASE EVIDENCE SHEET — COPY AND COMPLETE

```
DeployBoard release evidence — Chapter 8
Date (UTC):
Operator:
Docker Engine / Buildx versions:
Selected builder and supported platforms:
Source revision (commit or local-workshop):
App release tag:
Base image reference and index digest:
Resolved Python dependency list location:
Target platform(s):
Local image ID (single-platform):
OCI archive name and SHA-256 (if exported):
Registry image reference and index digest (if pushed):
CLI command used to verify /health:
Second clean build result and differences:
Decision: accepted / needs investigation
```

Make a clean-cache comparison without destroying other work

REBUILD THE SAME SOURCE WITHOUT USING THE BUILD CACHE

```
docker buildx build --builder db-release-builder \
  --no-cache --platform linux/amd64 --load \
  --tag deployboard-starter:0.3.0-clean .
docker image inspect deployboard-starter:0.3.0-clean \
  --format "{{.Id}}"
```

Compare the two image IDs, dependency output, build logs, and labels. If the image IDs differ, investigate mutable inputs such as the base tag, resolved dependencies, timestamps, and tooling rather than declaring the build broken. Matching image IDs are encouraging, but by themselves do not establish that all registry bytes, attestations, or external dependencies are identical.

SAFE EXIT docker rm -f db-ch08-api removes only this chapter's named test container. Keep the images and evidence if you want to use them in Chapter 9; never issue a broad docker system prune as part of a release-cleanup script.

You have now turned an optimized local build into a release you can describe, inspect, and carry across environments. In Chapter 9, the emphasis shifts from image identity to the communication boundaries between running containers.

Quick failure triage

If Buildx says the result remains in cache, choose --load, --push, or an explicit --output. If a requested platform has no matching manifest, inspect the base-image index and builder platforms. If a multi-platform --load fails, use an OCI archive or load just one platform. If digests differ, compare source, resolver output, base tags, build flags, and attached metadata. If ARM builds become unusually slow, investigate emulation rather than assuming an application defect.

PART III

CONNECT SERVICES AND KEEP THEIR DATA

Chapter 9 — Connect Containers with Docker Networking

Your DeployBoard starter image can answer an HTTP request, but a useful application cannot remain an island. Soon it will need a database and a worker. Before adding either one, you need to decide which processes should be able to talk, which names they should use, and which ports—if any—belong on your computer's network interfaces.

This chapter gives you a small, disposable networking range. You will observe two containers communicate by name, prove that container-internal ports do not require publication, briefly publish a web port only to your own computer, and leave PostgreSQL inaccessible through a host-published port. Every command is an investigation of a specific networking decision.

PREREQUISITE Docker Engine is running; the docker command works as shown in Chapter 1. You need an internet connection for the initial image pulls. All examples use Linux containers, which also run through Docker Desktop's Linux VM on Windows and macOS.

SHELL CONVENTION One-line docker commands can be used in Bash, macOS Terminal, Windows PowerShell, or WSL. Do not type a leading $ prompt. The host browser tests use http://127.0.0.1:8099. If another program already owns port 8099, stop the conflicting service or choose an unused host port consistently.

THE TEMPORARY LAB INVENTORY

Resource	Purpose	Data policy
net9-private	User-defined bridge for web and database	No important data
net9-guest	Separate network used to test isolation	Removed at the end
net9-web / net9-published	Unpublished web service / local-only web demo	Discardable
net9-db	Disposable PostgreSQL service	Test-only password and data

VERIFY DOCKER AND RETRIEVE THE THREE OFFICIAL LAB IMAGES

```
docker version
docker network ls
docker pull nginx:stable-alpine
docker pull busybox:1.36.1
docker pull postgres:17-alpine
```

The image downloads happen once per image version. Do not proceed if docker version cannot contact the Engine; diagnose that installation issue using Chapter 1 first. Tags used here are convenient for instruction, not immutable production pins. Chapter 8 explains how to record digests for release use.

9.1 Understand Bridge Networks and Service Isolation

A Docker bridge network connects containers managed by the same Docker daemon. On a Linux Engine, Docker implements bridging and network namespace interfaces in the host kernel. With Docker Desktop, those Linux networking mechanisms operate inside its managed Linux environment; the physical host is not identical to the Engine host. Either way, containers attached to the same user-defined bridge can connect to each other using container names.

The practical distinction is between a service address inside a network and an address available to your host or other devices. For example, an NGINX container can listen on port 80 without assigning port 80 on your laptop. A sibling container on the same bridge reaches it at http://net9-web:80. Only an explicit publication such as -p creates a host-side mapping.

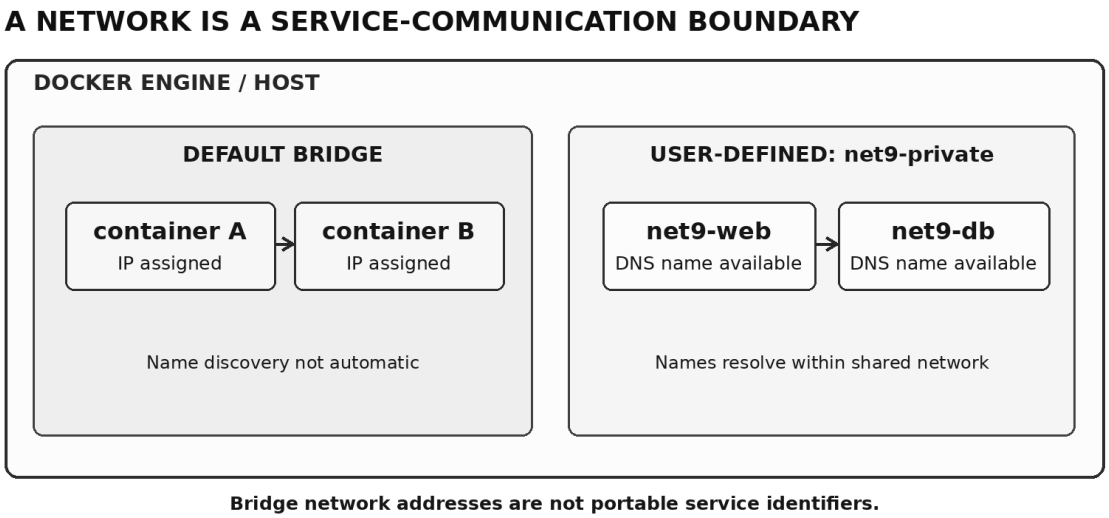


Figure 9.1 — Two network scopes. Prefer user-defined networks and stable names over memorized container IP addresses.

BOUNDARY, NOT MAGIC FIREWALL Separate bridge networks normally block direct container-to-container access, but do not treat a bridge name as an absolute security guarantee. Docker administrators, published ports, routing configuration, host services, and privileged processes affect the effective boundary. Review exposure on the real host in Chapter 27.

9.2 Compare Default and User-Defined Networks

Docker usually provides a default network named bridge. It is useful for basic examples, but its containers do not get automatic peer name resolution in the same way as a user-defined bridge. Named bridges provide a better project boundary and embedded DNS discovery. Neither choice changes the requirement that the service must actually be listening on the destination port.

Behavior	Default bridge	User-defined bridge
Peer discovery	Usually by IP unless legacy links are used	By container name or network alias
Scope	Unrelated containers may share it by default	You explicitly attach project services
Reconfiguration	More host-level coupling	Create and manage each bridge with the CLI
Recommended use	Legacy / quick experiments	Preferred for this book’s application services

OBSERVE THE DEFAULT NETWORK WITHOUT CHANGING IT

```
docker network ls
docker network inspect bridge
```

Read the reported Driver, Subnet, Gateway, and Containers entries. The addresses are assigned dynamically; do not copy example subnets into your application configuration. The system network named bridge should not be removed as part of this workshop.

9.3 Create and Inspect a Private Bridge Network

Create an explicitly named bridge so Docker assigns a suitable nonoverlapping subnet. The word “private” in its name expresses your intent; it does not automatically enable Docker’s separate --internal mode, nor does it remove all possible connections to the host. You will control exposure by attaching only the necessary containers and avoiding database port publication.

CREATE THE PROJECT NETWORK AND INSPECT ITS PROPERTIES

```
docker network create --driver bridge net9-private
docker network inspect net9-private --format "{{.Driver}}"
docker network inspect net9-private
```

Expected: Docker prints a network ID for the create operation, the abbreviated inspect prints bridge, and the full inspection shows the network’s assigned subnet and an initially empty Containers map. The exact network ID and subnet will differ from the book.

IF CREATION FAILS A duplicate name means this lab may already exist. Inspect net9-private before reusing it; never remove an unfamiliar network simply to make the example work. A failure to reach the daemon is an installation or permissions problem, not a networking design failure.

9.4 Resolve Container Names Through Docker DNS

Launch a named web container on the new bridge. NGINX listens on port 80 inside the container. No -p flag is used, so nothing is intentionally published to your host’s network interfaces. Next, use a short-lived BusyBox diagnostic container on the same network to look up the server name.

LAUNCH THE WEB SERVICE AND RESOLVE ITS NAME FROM A PEER

```
docker run -d --name net9-web --network net9-private nginx:stable-alpine
docker run --rm --network net9-private busybox:1.36.1 nslookup net9-web
```

Expected: nslookup displays a Docker-assigned address for net9-web. Your result may also display an embedded DNS server address such as 127.0.0.11 inside the querying container. That value refers to Docker DNS in the container's network namespace, not a public DNS service.

REQUEST THE RUNNING WEBSITE USING THE CONTAINER NAME

```
docker run --rm --network net9-private busybox:1.36.1 wget -qO- http://net9-web/
```

Expected: the default NGINX welcome HTML appears. The important evidence is that an independent container resolves net9-web and receives an HTTP response without using a fixed IP address. Service names are more stable than dynamically allocated IP addresses.

IF NAME RESOLUTION FAILS Check that net9-web is running and that both containers use --network net9-private. Do not try a fixed 172.x address first; that hides the misconfiguration you should learn to recognize.

9.5 Distinguish Exposed Ports from Published Ports

EXPOSE in a Dockerfile is image metadata describing an intended container port; it does not open the port on the laptop. The --expose option adds similar metadata when starting a container. Publishing is different: -p attaches an address and port on the Engine host to a container port. A same-network client can reach a listening port even if that port is not declared with EXPOSE and has never been published.

INSPECT THE IMAGE METADATA AND ACTUAL PORT BINDINGS

```
docker image inspect nginx:stable-alpine --format "{{json .Config.ExposedPorts}}"
docker port net9-web
docker container inspect net9-web --format "{{json .HostConfig.PortBindings}}"
```

Expected: the NGINX image identifies 80/tcp in ExposedPorts. docker port net9-web prints no mapping; PortBindings is empty or null. This is not a failure—the service remains reachable from the diagnostic container on net9-private.

Command or instruction	Actual effect
EXPOSE 80	Documents container port 80 in an image
docker run --expose 80	Marks a port as exposed at container creation

Command or instruction	Actual effect
docker run -p 8099:80	Publishes host port 8099 on default host interfaces
docker run -p 127.0.0.1:8099:80	Publishes the host's IPv4 loopback port 8099
Same-network request to net9-web:80	Uses internal communication, without host publication

A COMMON MISCONCEPTION Publishing a database port is not required for the API to connect to it over a shared Docker network. EXPOSE is not an access-control list and does not restrict what a process can listen on.

9.6 Use 127.0.0.1 Binding for Local-Only Services

For a browser demonstration, start a second disposable web container. Bind only IPv4 loopback on the host rather than writing -p 8099:80, which ordinarily publishes to all interfaces. This is appropriate for local learning, but it is not a substitute for host firewall validation or for applying authentication to services intended for public use.

CREATE A HOST-LOCAL PUBLISHED SERVICE

```
docker run -d --name net9-published --network net9-private -p 127.0.0.1:8099:80 nginx:stable-alpine
docker port net9-published
docker container ls --filter name=net9-published
```

Expected: docker port displays a mapping similar to 80/tcp -> 127.0.0.1:8099. Open http://127.0.0.1:8099 in a browser on the computer running Docker Desktop, or on the native Linux Docker host. The NGINX welcome page should appear. If you manage a remote Docker Engine, "localhost" refers to the remote Engine host, not automatically to your own computer.

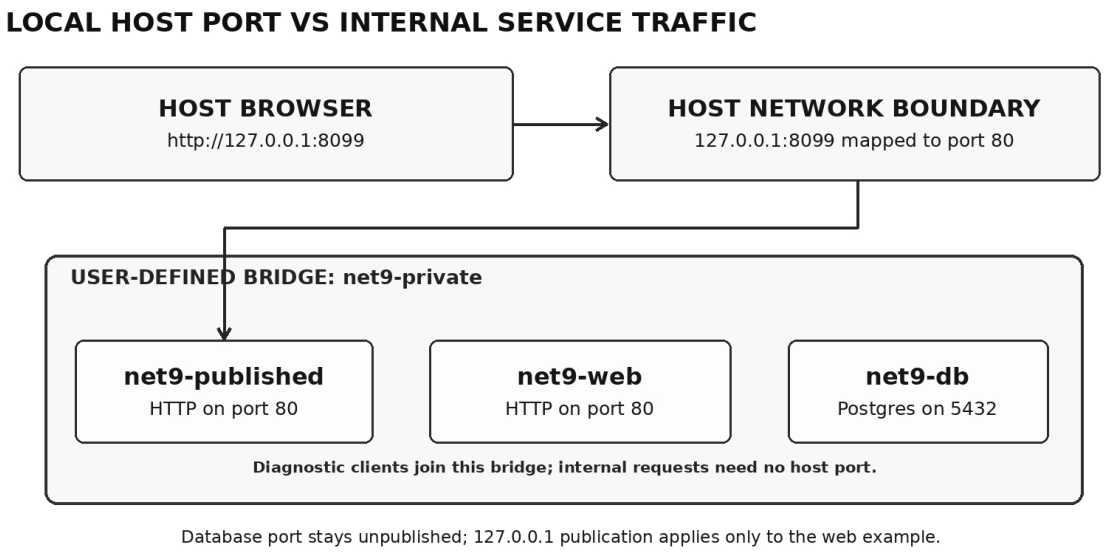


Figure 9.2 — The host loopback mapping is separate from same-network traffic; the database receives no host-published port.

Versions before Docker Engine 28.0.0 had a known condition in which other machines on the same layer-2 network could sometimes reach localhost-published ports. For a real deployment, verify engine version and exposure from another machine where authorized. Do not treat a successful local browser test as proof of external inaccessibility.

PORT CONFLICT If port 8099 is in use, choose another unused host port—for example 8109—and use that same number in the browser URL. The container port after the last colon remains 80.

9.7 Test Container-to-Container Communication

A network test must distinguish three things: name resolution, connection establishment, and application response. Name resolution alone does not prove a process is listening. A working HTTP response demonstrates both a reachable service port and a responding application. The one-off BusyBox client is removed after each test, so it does not leave extra running containers behind.

VERIFY BOTH DISCOVERY AND APPLICATION RESPONSE

```
docker run --rm --network net9-private busybox:1.36.1 nslookup net9-web
docker run --rm --network net9-private busybox:1.36.1 wget -qO- http://net9-web/
```

Create an unrelated bridge and attempt the same name lookup from outside the project network. The expected failure helps show the scope of automatic

Docker DNS. It is not a test of every possible routing path through the host.

DELIBERATELY QUERY FROM AN UNRELATED NETWORK — FAILURE EXPECTED

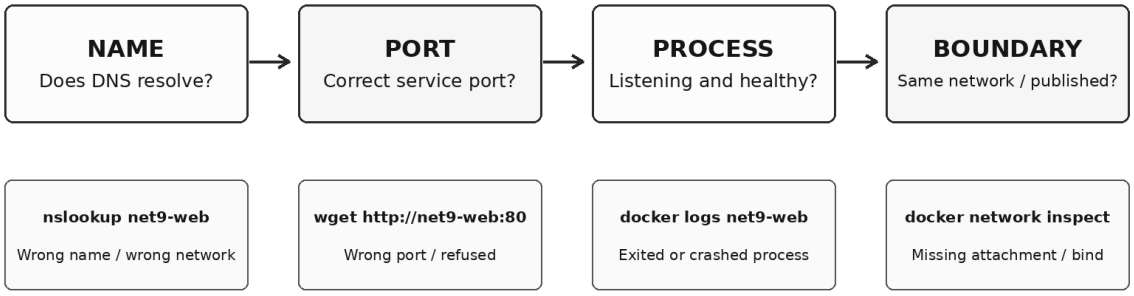
```
docker network create --driver bridge net9-guest
docker run --rm --network net9-guest busybox:1.36.1 nslookup net9-web
```

Expected: the nslookup command reports that net9-web cannot be resolved, and it may exit nonzero. That result is the purpose of the exercise. Do not "fix" the failure by publishing the service. Re-run the diagnostic container on net9-private and confirm that name resolution succeeds again.

9.8 Diagnose DNS, Routing, and Port Failures

A failed connection is an observation, not a diagnosis. Start with the smallest question: which address was requested? Then verify the network attachment, listening port, and process state. This order prevents the common mistake of adding -p, changing host firewall rules, or rebuilding images when the actual problem is only a mistyped name.

FROM A FAILED REQUEST TO A TESTABLE EXPLANATION



Collect evidence before changing flags or opening additional ports.

Figure 9.3 — Test name, port, process, and boundary in that order before changing configuration.

Symptom	Evidence to collect	Likely explanation
Name not found	docker network inspect + nslookup	Wrong name or network membership
Connection refused	Container logs + port in URL	Name resolves, process or port is wrong
HTTP error page	wget response + application logs	Reached service; application returned an error
Browser cannot open localhost	docker port + docker ps	Wrong host port, failed start, or remote daemon
Intermittent results	docker events + logs	Container restarts or unstable dependency

COLLECT EVIDENCE BEFORE CHANGING ANYTHING

```
docker container ls --filter name=net9-
docker network inspect net9-private
docker logs --tail 20 net9-web
docker port net9-published
```

Failure lab A: deliberately use http://net9-weeb/ in the one-off client; expect name resolution to fail. Failure lab B: use http://net9-web:8080/; expect an error because the default NGINX service listens on port 80, not 8080. Repair each request by using the correct name and port. The web service should not need to be rebuilt.

OBSERVE TWO EXPECTED FAILURES, THEN DEMONSTRATE A SUCCESSFUL REPAIR

```
docker run --rm --network net9-private busybox:1.36.1 wget -qO- http://net9-weeb/
docker run --rm --network net9-private busybox:1.36.1 wget -qO- http://net9-web:8080/
docker run --rm --network net9-private busybox:1.36.1 wget -qO- http://net9-web/
```

The first two commands deliberately return nonzero exit codes; execute them individually and read the error text before proceeding. Record the evidence that distinguishes “unknown hostname” from “host reachable but service port closed.”

9.9 Understand Host, None, and Overlay Network Boundaries

Docker provides drivers for substantially different network requirements. Most of this book runs a single host with user-defined bridges. Knowing the

boundaries of other drivers helps you avoid an unsafe or unnecessarily complicated configuration.

Mode	What changes	Use / caution
bridge	Per-container network stack with host-managed bridge	Best default for this chapter's services
host	Shares the Engine host network namespace	Port publishing does not apply; platform support differs
none	Only loopback networking inside the container	Useful for deliberate network isolation tests
overlay	Connects containers/services across Docker daemon hosts	Requires a suitable multi-host setup; not needed here

Host networking is not "faster bridge mode." It removes the normal network-namespace separation between the container and the Engine host, and -p mappings are ignored. On Docker Desktop its availability and behavior depend on Docker Desktop settings and version. Overlay networks are a different operational model, often connected with Swarm; Chapter 31 discusses when a single host is no longer enough.

DO NOT SWITCH MODES TO HIDE A BUG If a service name fails on a user-defined bridge, correct its network attachment and application port first. Moving it to host networking can conceal errors and increase exposure.

9.10 Limit Unnecessary Public Database Access

DeployBoard will later store deployment records in PostgreSQL. For now, create a disposable database container to practice the most important networking decision: other containers on the project bridge may reach port 5432 by name, but you do not publish that port to the host. The password below is intentionally a throwaway training value and can be seen in Docker container metadata. Never use it for an actual dataset or deployment.

CREATE AN UNPUBLISHED TRAINING DATABASE

```
docker run -d --name net9-db --network net9-private -e POSTGRES_PASSWORD=only-for-this-lab postgres:17-alpine
docker logs --tail 15 net9-db
docker port net9-db
```

Expected: PostgreSQL completes its initialization, and docker port net9-db displays no port mapping. On the initial startup, the logs may show

initialization messages and then “database system is ready to accept connections.” Exact wording and timing can vary. Do not publish -p 5432:5432 simply to make the next test work.

VERIFY DATABASE READINESS FROM ANOTHER CONTAINER

```
docker run --rm --network net9-private postgres:17-alpine pg_isready -h net9-db -U postgres
```

Expected after startup: net9-db:5432 - accepting connections (or equivalent). pg_isready checks the server’s connection readiness; it does not authenticate a user or prove that application queries will succeed. If it initially reports no response, wait for the database to finish its first initialization and repeat the command.

VERIFY NO PUBLISHED DATABASE PORT AND CORRECT NETWORK ATTACHMENT

```
docker container inspect net9-db --format "{{json .HostConfig.PortBindings}}"
docker network inspect net9-private
```

The official PostgreSQL image can create an anonymous data volume automatically. This lab does not teach persistence and intentionally stores no meaningful records. Chapter 10 introduces deliberate named volumes and data survival; Chapter 14 connects the real application and adds migrations. When cleanup removes net9-db with -v, only that disposable anonymous lab volume is targeted.

> **SECURITY BOUNDARY** Not publishing a database port removes one common exposure path, but is not by itself a complete database security policy. Require authentication, least-privilege credentials, restricted application networks, and separately validated host exposure in production.

9.11 Checkpoint: Build and Verify a Two-Service Network

Your final evidence should describe two durable running services on one user-defined bridge: the NGINX web service and the PostgreSQL database. A diagnostic container is launched only for each test. If you retained net9-published, it is an optional demonstration service and not required for this acceptance checkpoint.

Checkpoint commands — capture observed results

RUN THE ACCEPTANCE TESTS AND RECORD THE ACTUAL OUTPUT

```
docker container ls --filter name=net9-web --filter name=net9-db
docker network inspect net9-private
docker run --rm --network net9-private busybox:1.36.1 nslookup net9-web
docker run --rm --network net9-private busybox:1.36.1 wget -qO- http://net9-web/
docker run --rm --network net9-private postgres:17-alpine pg_isready -h net9-db -U postgres
docker port net9-db
```

Acceptance test	What counts as proof
Both services run	docker container ls shows net9-web and net9-db
Shared bridge membership	docker network inspect lists both names
Scoped name discovery	BusyBox resolves net9-web from net9-private
Working internal HTTP	BusyBox receives the NGINX welcome HTML
Database responds internally	pg_isready reports accepting connections
No database host publication	docker port net9-db has no mapping
Wrong network isolated by DNS	Name lookup from net9-guest fails as expected

Optional DeployBoard bridge test: if you still have the starter image from Chapter 8, run it on net9-private and query /health from BusyBox. This makes the service-address pattern concrete without introducing PostgreSQL application integration ahead of Chapter 14.

OPTIONAL — ONLY IF THE CHAPTER 8 IMAGE EXISTS LOCALLY

```
docker image inspect deployboard-starter:0.3.0
docker run -d --name net9-api --network net9-private deployboard-starter:0.3.0
docker run --rm --network net9-private busybox:1.36.1 wget -qO- http://net9-api:8000/health
```

Expected for the optional exercise: a JSON response containing a healthy status. If your Chapter 8 image uses a different tag, substitute its actual recorded tag. No new Dockerfile is needed. The API remains stateless here; the PostgreSQL integration belongs in a later part.

Deliberate, name-scoped cleanup

Keep the lab intact if you want to inspect it again. Otherwise, remove only the specifically named resources from this chapter. Do not run docker system prune or docker volume prune: those commands could affect unrelated projects. The database removal below is deliberately destructive to this temporary lab’s anonymous data volume; do not copy that -v pattern into a real database cleanup.

OPTIONAL: RUN ONLY IF YOU CREATED NET9-API

```
docker rm -f net9-api
```

OPTIONAL: RUN ONLY IF YOU CREATED NET9-PUBLISHED

```
docker rm -f net9-published
```

REMOVE THE REQUIRED WEB/DATABASE SERVICES AND LAB NETWORKS

```
docker rm -f net9-web
docker rm -f -v net9-db
docker network rm net9-guest
docker network rm net9-private
```

If an optional or guest resource was never created, its removal command may say “No such container” or “network not found”; that is harmless. If a network reports active endpoints, inspect those endpoints before deleting anything else. At this point you can explain how names, network attachments, internal ports, published ports, and engine-host boundaries fit together.

WHAT COMES NEXT Chapter 10 adds intentional persistent storage. You will replace the disposable database-data assumption with named volumes and evidence that a container can be recreated without losing records.

Chapter 10 — Keep Data Safe with Volumes and Bind Mounts

A container can be replaced in seconds. Its data cannot. In this workshop you will separate those lifetimes, inspect where files live, and prove that a PostgreSQL database survives container replacement. Then you will make a logical backup and restore it into an independent database—the first recovery proof in the DeployBoard journey.

WORKSHOP CONTRACT

You will use	You will produce	What counts as proof
Docker CLI; BusyBox and NGINX images	A persistent note and a read-only host mount	Data is visible in a second container
Official PostgreSQL 17 image	A database backed by a named volume	Two rows remain after container replacement
pg_dump and psql	A plain SQL backup and separate restore	Restored database contains the same records

SCOPE This is an isolated storage lab—not the DeployBoard application database. Chapter 14 connects PostgreSQL to the actual API, introduces migrations, and manages application credentials. Use a terminal running Bash (Linux/macOS/WSL) or PowerShell (Windows).

10.1 Explain Ephemeral Container Writable Layers

An image provides a starting filesystem; a running container gains a writable layer above it. That layer belongs to the container object. Stopping and starting the same container normally preserves it, but removing the container removes its writable layer. A newly created container does not inherit files written only into the previous container.

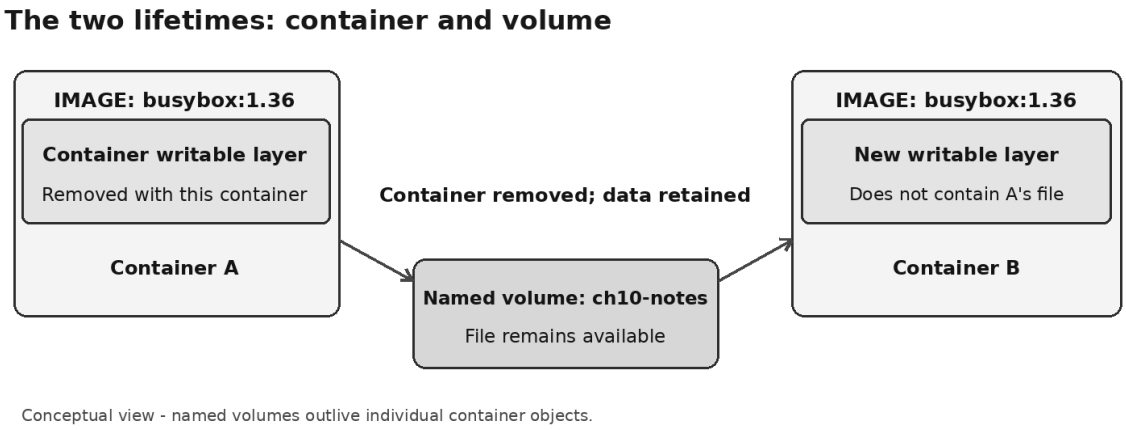


Figure 10.1 Container-owned writable state versus Docker-managed volume state.

Run this controlled experiment. It writes a file inside a disposable container and reads it back before that container is removed.

Terminal — create a file in a container

```
docker run --name ch10-ephemeral busybox:1.36 sh -c "echo temporary > /tmp/note.txt; cat /tmp/note.txt"
```

Expected output: temporary. The container is now stopped, but its writable layer is still associated with that container. Read the same file once more by copying it to your working directory:

```
docker cp ch10-ephemeral:/tmp/note.txt ./ch10-observed.txt
docker rm ch10-ephemeral
```

Open ch10-observed.txt with a text editor. It contains temporary because docker cp moved an independent copy onto your host. After docker rm, the container-only original is gone. A new BusyBox container does not contain /tmp/note.txt:

```
docker run --rm busybox:1.36 sh -c "test ! -f /tmp/note.txt && echo fresh"
```

Expected output: fresh. Later chapters will put configuration and caches into images, while durable application data moves into named volumes or another external data service.

10.2 Create, Inspect, and Reuse Named Volumes

A named volume is a persistent storage object managed by the Docker engine. The volume can be attached to different containers at the same target path; deleting one of those containers does not delete the named volume.

Terminal — create and inspect your first volume

```
docker volume create ch10-notes
docker volume inspect ch10-notes
```

Inspect reports fields such as Name, Driver, Mountpoint, and Scope. On Docker Desktop the mountpoint is inside Docker's Linux environment; it is not a normal Windows or macOS folder to edit manually. Write a file into the volume:

```
docker run --rm --mount "type=volume,source=ch10-notes,target=/notes" busybox:1.36 sh -c "printf 'Persistent note\n' > /notes/message.txt"
```

The writing container exits and is automatically removed. Start a new container with the same volume:

```
docker run --rm --mount "type=volume,source=ch10-notes,target=/notes" busybox:1.36 cat /notes/message.txt
```

Expected output: Persistent note. The content remained because the named volume, not the disposable container, owned the file. Docker allows multiple mounts of the same volume, but safe concurrent writes still depend on the applications involved; a named volume is not a general-purpose database locking system.

10.3 Choose Between Volumes and Bind Mounts

Both mechanisms map external storage into a container. The right choice depends on who should manage the source path. A named volume is usually the simplest default for persistent service data; a bind mount is useful when you deliberately want to edit or inspect the source from your host.

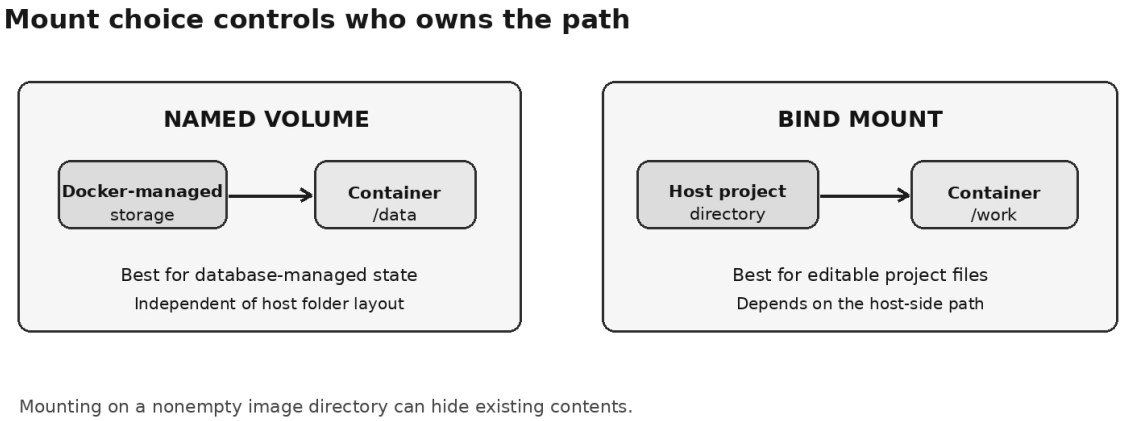


Figure 10.2 Named volumes and bind mounts connect different owners to a container path.

Decision	Named volume	Bind mount

Decision	Named volume	Bind mount
Where is the source?	Docker-managed storage	Exact host file or directory
Common use	Databases; durable service state	Source code; configuration; local fixtures
Portability	Independent of host path layout	Requires a suitable host path
Main hazard	Deleting an unused volume	Exposing or overwriting host files

DO NOT CONFUSE A volume is persistence, not a backup. If an application deletes its own records or corrupts data inside a volume, that damage persists too. Section 10.10 creates an independent recovery artifact.

10.4 Understand Mount Paths and Initialization Behavior

Mounting external storage over a container directory changes what the process sees at that directory. A bind mount hides the existing image files beneath the target while attached. An empty named volume typically receives a one-time copy of pre-existing image files at its target, unless copy-up is disabled. This can be useful for configuration but surprising if you expect an empty directory.

Observe volume initialization using NGINX's packaged default website:

```
docker volume create ch10-web-seed
docker run --rm --mount "type=volume,source=ch10-web-seed,target=/usr/share/nginx/html"
nginx:stable-alpine ls /usr/share/nginx/html
```

Expected result: the default website files, typically including index.html and 50x.html. The previously empty named volume receives copies from the image when first mounted. Confirm that the files now belong to the volume by mounting it with BusyBox:

```
docker run --rm --mount "type=volume,source=ch10-web-seed,target=/pages" busybox:1.36 ls
/pages
```

The copy occurs on initial mounting of an empty volume; later mounts do not overwrite existing volume data with newer image files. The --mount volume-nocopy option disables initial copying. Do not mount a new empty host directory over a prepopulated application directory and expect its image contents to remain visible.

10.5 Share Files Safely Across Host and Container

Bind mounts are useful when you own the host-side file and want a container to read it. Create a new folder called chapter10-lab, open your terminal in that folder, and create the following plain-text file. The commands in this section assume the terminal's current directory is that folder.

Complete file listing

```
HOST FILE: bind-note.txt
========================
Edited on the host; read inside a container.
```

Now run this command in Bash or PowerShell. In both shells, $PWD identifies the current directory. Do not use the unchanged command in Windows Command Prompt (cmd.exe), where path expansion differs.

```
docker run --rm --mount "type=bind,source=$PWD,target=/work,readonly" busybox:1.36 cat /work/bind-note.txt
```

Expected output is the one-line file content. Change the text using a host editor, rerun the command, and observe the updated content. You did not rebuild an image; the container read the host file directly.

The readonly setting prevents container-side writes through that mount. This is preferable for configuration and fixtures that the container does not need to modify.

> **SAFETY** Do not casually bind-mount your home folder, SSH keys, Docker socket, or system directories into containers. A writable bind mount can change host files. With --mount, a nonexistent source path produces an error instead of silently creating the wrong folder.

10.6 Recognize Ownership and Permissions Problems

A bind mount exposes real host-side permission behavior. On Linux, access decisions depend on numeric user and group IDs; the user name displayed inside an image may have no equivalent on the host. Windows and macOS add Docker Desktop file-sharing and permission translation layers.

Inspect the user in a temporary container:

```
docker run --rm busybox:1.36 id
docker run --rm --user 10001:10001 busybox:1.36 id
```

In the second command the process runs as UID and GID 10001. This does not grant permission to every mounted file. A file readable by the default container user might produce Permission denied for a non-root user.

When a mount fails, inspect in this order

First confirm the source file exists at the path passed to Docker. Next check whether the mount is readonly. On Linux, inspect file and directory ownership and the execute permission on parent directories. Then compare the process UID/GID with the permissions on that path. On Docker Desktop, confirm that the host folder is shared and accessible to Docker's Linux environment.

Avoid solving a permissions problem with chmod 777 or by making the entire container privileged. Give only the required user access to the required path.

10.7 Account for Windows, macOS, and Linux Differences

The Docker command is mostly portable, but the path comes from your host shell and operating system. Named volumes behave consistently enough for this book's database lab; bind-mount performance, path syntax, filesystem events, and ownership behavior vary.

Learning environment	Recommended practice	What can differ
Windows + Docker Desktop	PowerShell or WSL 2 shell; use $PWD in the matching shell	Windows drive paths, sharing permissions, case handling
WSL 2 (Ubuntu)	Keep active project files in the Linux filesystem	/mnt/c bind mounts may be slower; watcher behavior
macOS + Docker Desktop	Use a shared project directory	VM-backed volume paths; host permissions
Native Linux Engine	Use absolute Linux paths and ordinary UID/GID checks	Direct host ownership and daemon permissions

For a Windows developer using WSL 2, placing frequently accessed source code within the WSL Linux filesystem generally improves bind-mount performance compared with a project under /mnt/c. A named volume is the simpler default for PostgreSQL data across the supported platforms.

10.8 Run a Standalone PostgreSQL Container with Durable Storage

Now you will create the database that makes persistence measurable. This is intentionally a standalone lab database: it is not yet connected to DeployBoard, and its port is not published on the host. The only access used in this chapter is docker exec.

Inside chapter10-lab, create pg10.env exactly as shown. These values are for a disposable local learning database. They are not production credentials or a production secret-management pattern.

Complete file: pg10.env

```
POSTGRES_USER=labuser
POSTGRES_PASSWORD=local-demo-only-change-me
POSTGRES_DB=labdb
```

Keep the file out of any public repository. The container's environment can be inspected by users with appropriate Docker access; a stronger secret-handling method is introduced in Chapter 19.

Terminal — start a private PostgreSQL 17 instance

```
docker pull postgres:17
docker volume create ch10-db-data
docker run -d --name ch10-postgres --env-file pg10.env --mount "type=volume,source=ch10-db-data,target=/var/lib/postgresql/data" postgres:17
```

Do not append -p 5432:5432. PostgreSQL is reachable only through Docker facilities in this lab, not through a new published host port. Give the service time to initialize, then check readiness; if it reports no response, repeat after initialization has finished:

```
docker exec ch10-postgres pg_isready -U labuser -d labdb
docker logs --tail 20 ch10-postgres
```

When ready, pg_isready reports accepting connections. Create seed.sql in the same folder using the complete listing below.

Complete file: seed.sql

```
CREATE TABLE IF NOT EXISTS lab_notes (
   id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
   note text NOT NULL
);

INSERT INTO lab_notes (note) VALUES
   ('Survives container replacement'),
   ('Included in the logical backup');

SELECT id, note FROM lab_notes ORDER BY id;

docker cp ./seed.sql ch10-postgres:/tmp/seed.sql
docker exec ch10-postgres psql -X -v ON_ERROR_STOP=1 -U labuser -d labdb -f /tmp/seed.sql
```

Expected result: CREATE TABLE, INSERT 0 2, and two rows displayed in order. You have now created real records inside PostgreSQL's data directory, which is backed by ch10-db-data.

VERSION BOUNDARY This lab intentionally uses postgres:17 and mounts /var/lib/postgresql/data. Official PostgreSQL 18+ Docker images use a changed, version-specific PGDATA layout and a volume at /var/lib/postgresql. Do not replace the tag with latest or silently reuse the same mount path for a newer major version.

10.9 Recreate PostgreSQL Without Losing Test Records

A restart proves less than you think: it keeps the same container object. The stronger test is to remove that object and start a new one pointing at the same named volume.

Terminal — remove the container, not its data volume

```
docker stop ch10-postgres
docker rm ch10-postgres
docker volume inspect ch10-db-data
```

You should still see the named volume in docker volume inspect. Create a replacement from the same PostgreSQL major version, the same initialization settings, and the same existing volume:

```
docker run -d --name ch10-postgres --env-file pg10.env --mount "type=volume,source=ch10-db-data,target=/var/lib/postgresql/data" postgres:17

docker exec ch10-postgres pg_isready -U labuser -d labdb
docker exec ch10-postgres psql -X -U labuser -d labdb -c "SELECT COUNT(*) AS saved_notes FROM lab_notes;"
```

Expected result: saved_notes = 2. The records were not rebuilt by the image or recreated from seed.sql. They survived because they were stored in the named volume. The POSTGRES_USER, POSTGRES_PASSWORD, and POSTGRES_DB initialization settings apply when the data directory is first initialized; changing them later does not automatically recreate an existing database cluster.

10.10 Export a Basic Development Backup

Container replacement proves persistence but not recovery from accidental data deletion. A logical PostgreSQL dump gives you a separately stored SQL artifact. The safest first test is to restore that artifact into a clean database, without changing the original one.

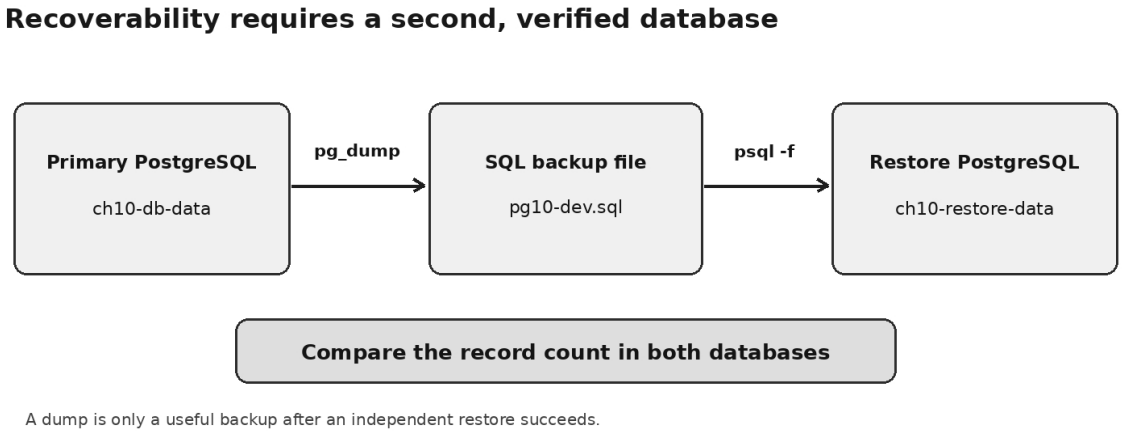


Figure 10.3 Dump, copy to the host, restore into a separate volume, then verify.

Generate a plain SQL dump within the running PostgreSQL container and copy it into chapter10-lab:

Terminal — produce the backup file

```
docker exec ch10-postgres pg_dump -U labuser -d labdb --no-owner --no-privileges -f /tmp/pg10-dev.sql
docker cp ch10-postgres:/tmp/pg10-dev.sql ./pg10-dev.sql
```

Open pg10-dev.sql in a text editor. It should contain table definitions and COPY or INSERT data representing the two notes. Do not modify it. A backup file may contain sensitive application data; protect it accordingly and never commit realistic production dumps to a public repository.

Prove the backup by creating an independent database on a fresh named volume. Use the same major PostgreSQL version and this chapter’s lab env file:

```
docker volume create ch10-restore-data
docker run -d --name ch10-restore --env-file pg10.env --mount "type=volume,source=ch10-restore-data,target=/var/lib/postgresql/data" postgres:17
```

Wait until the second database accepts connections, then copy the dump into it and load the SQL file. The -v ON_ERROR_STOP=1 option makes psql exit on an SQL error rather than quietly continuing past it.

```
docker exec ch10-restore pg_isready -U labuser -d labdb
docker cp ./pg10-dev.sql ch10-restore:/tmp/pg10-dev.sql
docker exec ch10-restore psql -X -v ON_ERROR_STOP=1 -U labuser -d labdb -f /tmp/pg10-dev.sql

docker exec ch10-restore psql -X -U labuser -d labdb -c "SELECT COUNT(*) AS saved_notes FROM lab_notes;"
```

Expected result: saved_notes = 2 in the restored database. The file is now a tested development backup. For a real system, backup scope also includes roles, other databases, permissions, encryption, retention, and recovery objectives. A single-database pg_dump is not a full PostgreSQL cluster backup; those details receive systematic treatment in Chapter 29.

10.11 Checkpoint: Prove Persistence and Recovery

Treat the final checkpoint as an operator handoff. Someone else should be able to reproduce the evidence from your commands without relying on an unexplained host directory or data hidden inside a disposable container.

Release gate — all outcomes must be observable

- □ Named volume ch10-notes contains message.txt when mounted by a second container.
- □ Bind-mounted bind-note.txt reflects a host edit while remaining readonly in the container.
- □ Container ch10-postgres uses the named volume ch10-db-data at the PostgreSQL 17 data directory.
- □ The original PostgreSQL database still contains two notes after its container was removed and recreated.
- □ pg10-dev.sql exists on the host and was created by pg_dump.
- □ A separate PostgreSQL container on ch10-restore-data successfully restored the dump.

- □ The restored database contains two notes; no database port was published to the host.

If the lab fails, make a specific diagnosis

Symptom	Likely explanation	First check
Bind mount path fails	Source path missing or folder sharing blocked	Confirm $PWD and file existence
Database initializes unexpectedly	Wrong volume name or mount target	Inspect named volume and exact PostgreSQL 17 path
psql reports relation does not exist	seed.sql was not loaded or wrong database	Inspect command exit and -d labdb
Restore fails	Destination not ready or existing objects conflict	Use clean restore volume; check psql error text

Close the lab without accidental data loss

Remove only the temporary lab containers. Leave the named volumes in place until you have retained the backup and finished reviewing the checkpoint:

```
docker rm -f ch10-postgres ch10-restore
docker volume ls
```

If—and only if—you intentionally want to discard every storage artifact created in this chapter, you can delete these specifically named volumes after verifying that no containers still use them:

```
docker volume rm ch10-notes ch10-web-seed ch10-db-data ch10-restore-data
```

Do not substitute docker volume prune or docker system prune --volumes for this targeted cleanup. Those commands may delete unrelated data. Keep pg10-dev.sql outside Docker if you want the independent recovery record; delete the lab password file if it is no longer needed.

What changes next

You have now proved that a Docker-managed database volume survives container replacement and that a logical backup can be restored elsewhere. In Chapter 11, you will describe services, volumes, and networks declaratively using Docker Compose. In Chapter 14, this standalone proof becomes a real DeployBoard database connected to application code and migrations.

Chapter 11 — Build Your First Multi-Container Compose Stack

THE DELIVERY PROBLEM

A web container is running, and so is PostgreSQL. They have different names, different lifecycles, and different storage requirements. You could type a long set of docker run commands every time, but those commands are easy to forget and difficult to review. In this chapter, you make the two services a single, inspectable Compose application.

You will package a small web demonstration—not the full DeployBoard API—so this chapter teaches Compose itself. Chapters 13–16 will give the real product its domain model, database integration, workers, and application-level checks. For now, the web service only checks whether the database TCP port can be reached. That is enough to make network discovery and failure behavior visible without inventing business logic.

Start here	You will finish with
Docker Engine running; docker compose version works	A portable compose.yaml, local web image, two services and a named volume
Terminal in a new, disposable folder	Proof of service-name DNS, database readiness and retained test rows
An editor and internet for the initial image pulls	Clear evidence of what Compose does—and does not—recover

SAFETY BOUNDARY Use only the chapter directory docklab11. The example database password is public demonstration data, not a real credential. Never deploy this exact Compose file on an internet-facing server.

11.1 Explain the Compose Application Model

Compose describes a multi-container application in one declarative YAML model. A service states which image runs, how it is configured, which networks it joins, and which mounts it uses. The Compose project supplies a shared identity to containers, networks, and volumes so you can operate the application as a unit.

You still use the Docker Engine underneath. Compose does not replace images, containers, networking, or the daemon. It coordinates their creation and makes the desired configuration explicit. It also does not magically provide high availability, database replication, or an ongoing reconciliation controller.

Compose term	What it means here
Project	docklab11 is the namespace for resources created from this file.
Service	web and db are named, repeatable container definitions.
Network	edge connects web to an external-facing endpoint; data connects web and db.
Volume	db_data stores PostgreSQL files separately from the database container.
Health check	A command that provides the db service’s health status.

Operating principle: read the YAML like a system diagram. When a service cannot connect, inspect the service name, network membership, port, process, and health status before changing configuration.

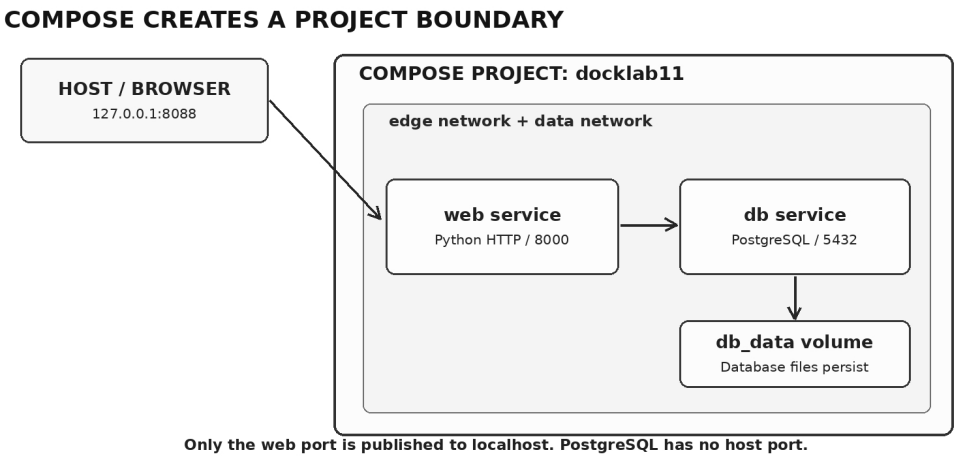


Figure 11.1 — Two services, two networks, one project, and one persistent volume.

11.2 Create the First compose.yaml File

Start in a new folder instead of modifying the Chapter 6 or Chapter 10 files. Use any text editor and save the filenames exactly. Compose recognizes compose.yaml in the current project directory; the web image will be built from the web subfolder.

BASH OR POWERSHELL — CREATE THE PROJECT FOLDERS

```
mkdir docklab11
cd docklab11
mkdir web
```

FINAL PROJECT TREE (CREATE THESE FILES IN YOUR EDITOR)

```
docklab11/
  compose.yaml
  web/
    Dockerfile
    .dockerignore
    app.py
```

The tiny web service

This self-contained Python program serves JSON on port 8000. It opens a short-lived TCP connection to the database hostname for every request, then reports reachable or unreachable. It does not authenticate to PostgreSQL or read records; those are different checks.

The next two listings are consecutive portions of the same web/app.py file. Paste them in order into one file; the labels are not part of the code.

FILE: WEB/APP.PY — LISTING 1 OF 2 (IMPORTS AND CONFIGURATION)

```
"""Chapter 11: intentionally small Compose networking demonstration.

This is NOT the full DeployBoard API from Chapter 13.
"""
import json
import os
import socket
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

DB_HOST = os.environ.get("DB_HOST", "db")
DB_PORT = int(os.environ.get("DB_PORT", "5432"))
```

FILE: WEB/APP.PY — LISTING 2 OF 2 (HANDLER AND SERVER)

```
class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path not in ("/", "/health"):
            self.send_error(404, "Use / or /health")
            return

            try:
            with socket.create_connection((DB_HOST, DB_PORT), timeout=2):
                database_tcp = "reachable"
        except OSError:
            database_tcp = "unreachable"

            payload = {
            "service": "docklab11-web",
            "database_host": DB_HOST,
            "database_tcp": database_tcp,
        }
        body = json.dumps(payload, indent=2).encode("utf-8")
        self.send_response(200 if database_tcp == "reachable" else 503)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)

if __name__ == "__main__":
    server = ThreadingHTTPServer(("0.0.0.0", 8000), Handler)
    print("docklab11-web listening on 8000", flush=True)
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.server_close()
```

The root path and /health share a deliberately simple response. When the database port is unavailable, the endpoint returns HTTP 503. That provides a repeatable failure you can observe in Section 11.8.

FILE: WEB/DOCKERFILE

```
FROM python:3.13-slim
WORKDIR /app
COPY app.py /app/app.py
USER 65532:65532
EXPOSE 8000
CMD ["python", "-u", "/app/app.py"]
```

FILE: WEB/.DOCKERIGNORE

```
__pycache__/
*.pyc
.env
.git/
```

This build has no third-party Python package dependencies. The runtime user is non-root, and the application listens inside its container on 0.0.0.0:8000 so Docker's host-port mapping can forward requests to it.

11.3 Define a Demonstration Web Service and Database Service

Now express both processes in one Compose file. The web service is built locally and given an image tag. The database uses the official PostgreSQL 17 image. That major version is deliberate: in the official PostgreSQL 17 image, /var/lib/postgresql/data is the correct persistent mount target. PostgreSQL 18 and later use a different default layout.

FILE: COMPOSE.YAML — COMPLETE AND CANONICAL LISTING

```
name: docklab11

services:
  web:
    build:
      context: ./web
    image: docklab11-web:1.0
    pull_policy: build
    environment:
      DB_HOST: db
      DB_PORT: "5432"
    ports:
      - "127.0.0.1:8088:8000"
    depends_on:
      db:
        condition: service_healthy
    networks:
      - edge
      - data

  db:
    image: postgres:17
    environment:
      POSTGRES_USER: demo
      POSTGRES_PASSWORD: demo-only-change-me
      POSTGRES_DB: demo
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U demo -d demo"]
      interval: 3s
      timeout: 3s
      retries: 10
      start_period: 10s
    networks:
      - data

networks:
  edge: {}
  data:
    internal: true

volumes:
  db_data: {}
```

Read three important boundaries before starting: only the web service publishes a host port; db is discoverable only through Docker networks; and the sample password is included strictly to make this offline lab reproducible. A public deployment needs secrets, access controls, and a separate production configuration.

VERSION SCOPE This chapter uses the modern docker compose command (Compose v2). It does not use the older docker-compose standalone command. Confirm your local capabilities with docker compose version before proceeding.

11.4 Configure Named Networks and Volumes

The web service attaches to edge and data; db attaches only to data. Since both services share data, the web service can resolve the database by the name db. The data network is internal to the Docker network model; its purpose here is to keep the database away from ordinary outbound connectivity. Do not mistake this single setting for a complete security boundary against privileged host users.

A named volume is more durable than a container writable layer. Compose creates docklab11_db_data and mounts it at the location where PostgreSQL 17 stores its cluster files. A normal compose down removes application containers and project networks but preserves the volume.

Resource	Declared key	Likely Docker resource name
Front-end bridge	edge	docklab11_edge
Private data bridge	data	docklab11_data
PostgreSQL storage	db_data	docklab11_db_data

The resource names above follow the typical project-prefixed convention. Always confirm the actual names with docker network ls and docker volume ls rather than relying on the examples as fixed IDs.

11.5 Distinguish build from image References

The web service declares build because you own its Dockerfile and source. It also declares image so the resulting build has a readable name: docklab11-web:1.0. pull_policy: build makes the intended local build explicit instead of leaving the pull-versus-build precedence ambiguous. The db service uses image alone because this lab consumes the official upstream PostgreSQL image.

FROM DOCKLAB11 — VALIDATE AND BUILD THE WEB IMAGE

```
docker compose version
docker compose config --quiet
docker compose build web
```

INSPECT THE IMAGE TAG AFTER THE BUILD

```
docker image ls docklab11-web
```

Build results and hashes vary. You should see a locally built image named docklab11-web with tag 1.0. BuildKit may reuse cached layers, and the first base-image download can take longer than later builds.

RELEASE BOUNDARY Tagging an image in Compose does not make that tag immutable. Chapter 8 explains why release digests matter; for this local chapter, the readable image tag is enough to distinguish the service image from the database image.

11.6 Use Service Names Instead of Fixed IP Addresses

Container addresses may change when Compose replaces a container. The service name remains stable within a shared Compose network. Your web program uses DB_HOST=db, not an invented IP address or localhost. Inside web, localhost refers to web itself—not the database container.

LAUNCH SERVICES AND RESOLVE THE DB SERVICE NAME

```
docker compose up -d
docker compose ps
docker compose exec -T web python -c "import socket; print(socket.gethostbyname('db'))"
```

The DNS command should print a bridge-network IP address; the exact address differs on each machine. Check the response from a host browser at http://127.0.0.1:8088. You can also use curl from Bash/macOS or curl.exe from PowerShell.

BASH OR MACOS — REQUEST THE ENDPOINT

```
curl http://127.0.0.1:8088/
```

POWERSHELL — USE CURL.EXE TO AVOID ALIAS DIFFERENCES

```
curl.exe http://127.0.0.1:8088/
```

REPRESENTATIVE JSON BODY AFTER BOTH SERVICES ARE HEALTHY

```
{
  "service": "docklab11-web",
  "database_host": "db",
  "database_tcp": "reachable"
}
```

The reported TCP connection proves that the web container can reach the PostgreSQL listener through the Compose network. It does not prove SQL permissions, database schema availability, or the correctness of an application's business operations.

11.7 Understand depends_on and Health-Based Readiness

A database container can be running before PostgreSQL is ready to accept connections. A short depends_on list only establishes startup order; it does not by itself wait for readiness. The long form in this lab waits for db to pass its defined health check before starting web.

OBSERVE SERVICE HEALTH AND READINESS

```
docker compose ps
docker compose logs --tail=25 db
docker compose exec -T db pg_isready -U demo -d demo
```

Expected result: the db service eventually shows healthy in docker compose ps, and pg_isready reports accepting connections. Startup duration differs by hardware and whether the volume is new. pg_isready tests server readiness, not whether your application account has completed an authenticated SQL query.

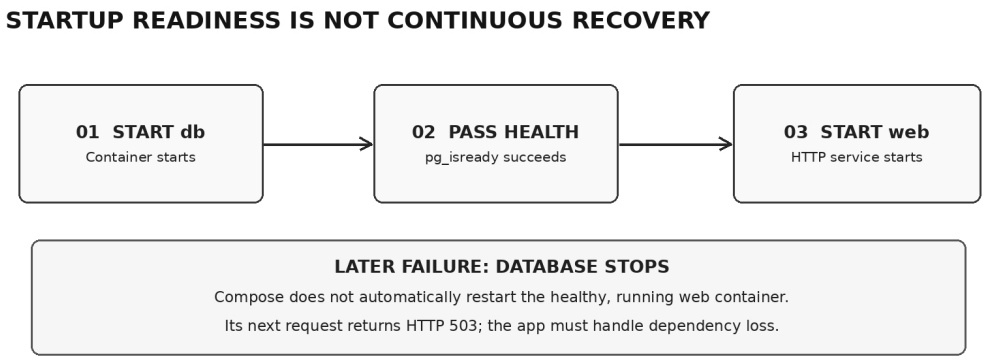


Figure 11.2 — Health-gated startup is one-time ordering, not a recovery controller.

Readiness may also be checked by the application when handling requests. Our simple web service tests the database port on each request, which makes database loss visible even though Compose itself does not continuously orchestrate dependency health.

11.8 Distinguish Startup Ordering from Ongoing Failure Recovery

Now prove a limitation instead of memorizing it. Stop db after the stack is already running. The web container normally stays running because depends_on does not function as a continuous supervisor for the entire application. Its next HTTP request should report an unavailable database and return 503.

STOP ONLY THE DATABASE SERVICE; INSPECT BOTH CONTAINERS

```
docker compose stop db
docker compose ps -a
```

BASH OR MACOS — SEE THE HTTP RESPONSE STATUS

```
curl -i http://127.0.0.1:8088/
```

POWERSHELL — EQUIVALENT REQUEST

```
curl.exe -i http://127.0.0.1:8088/
```

REPRESENTATIVE FAILURE EVIDENCE; RESPONSE HEADERS VARY

```
HTTP/1.0 503 Service Unavailable
...
"database_tcp": "unreachable"
```

The web process remains alive, but a request requiring database connectivity cannot succeed. Restarting an unhealthy process, reconnecting to a replaced database, retrying failed work, and alerting on persistent dependency loss are application/operations decisions —not automatic consequences of depends_on.

RESTART DB; WAIT FOR HEALTHY, THEN RETEST THE WEB ENDPOINT

```
docker compose start db
docker compose ps
docker compose logs --tail=15 web
```

Once PostgreSQL becomes reachable again, refresh http://127.0.0.1:8088 and expect database_tcp to become reachable. This example recovers at request time without a web

restart because it opens a fresh socket for every request. A long-lived database client may require a reconnection strategy.

DO NOT CONFUSE TWO RESTART CONTROLS The optional depends_on: db: restart: true applies to certain explicit Compose-driven updates or restarts of the dependency. It does not mean Docker will automatically restart web whenever db crashes. A service-level restart policy also does not automatically repair an unhealthy but still-running application.

11.9 Build, Start, Inspect, and Stop the Stack

Compose provides an application-level operating interface. Run these commands from docklab11 so Compose loads the intended file. Use service keys (web and db), not generated container names, for normal day-to-day operations.

Task	Command
Build the local service	docker compose build web
Create / start in background	docker compose up -d
See service state	docker compose ps
Inspect recent service logs	docker compose logs --tail=30 web
Run a SQL client inside db	docker compose exec -T db psql -U demo -d demo
Stop containers, retain them	docker compose stop
Restart stopped containers	docker compose start
Remove stack, retain volume	docker compose down

Let’s make persistence observable. Insert one harmless record through the PostgreSQL client inside the database container; no database host port is necessary.

CREATE A REPEATABLE ONE-ROW DEMONSTRATION TABLE

```
docker compose exec -T db psql -U demo -d demo -c "CREATE TABLE IF NOT EXISTS compose_probe (id
integer PRIMARY KEY, note text NOT NULL);"
```

INSERT THE RECORD (SAFE TO RUN MORE THAN ONCE)

```
docker compose exec -T db psql -U demo -d demo -c "INSERT INTO compose_probe (id, note) VALUES (1,
'retained across down') ON CONFLICT (id) DO UPDATE SET note = EXCLUDED.note;"
```

CONFIRM THE STORED RECORD

```
docker compose exec -T db psql -U demo -d demo -tAc "SELECT note FROM compose_probe WHERE id = 1;"
```

EXPECTED QUERY RESULT (WHITESPACE MAY VARY)

```
retained across down
```

Next, remove the application containers and project networks while retaining the volume, then recreate the stack. This is not a substitute for a backup; it is only a proof of volume persistence.

RECREATE THE STACK, THEN READ THE RETAINED RECORD

```
docker compose down
docker compose up -d
docker compose exec -T db psql -U demo -d demo -tAc "SELECT note FROM compose_probe WHERE id = 1;"
```

You should retrieve the same text. If the table disappears, inspect the named volume, the declared mount path, and whether you accidentally used down --volumes. The volume demonstration in Chapter 10 provides the storage model; this Compose exercise demonstrates the same protection at project level.

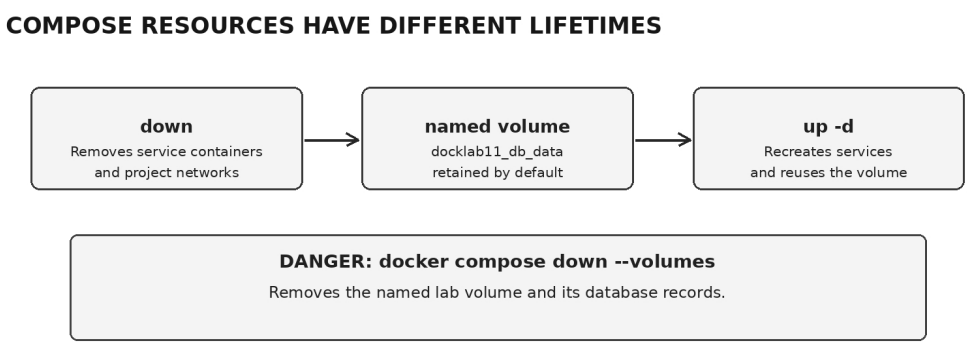


Figure 11.3 — A normal Compose teardown is not the same as deleting a volume.

11.10 Validate Configuration with docker compose config

Treat the rendered configuration as the actual plan Compose will execute. docker compose config expands the YAML model, normalizes fields, and checks its validity. This is useful before a service is created, especially when variables, overrides, or multiple files are introduced later in Chapter 12.

VALIDATE AND INSPECT THE RESOLVED APPLICATION MODEL

```
docker compose config --quiet
docker compose config
```

Find both services, the localhost-only port mapping, depends_on with service_healthy, the two networks, and the db_data volume. If the command fails, repair YAML indentation or mistyped service keys before running up. The official YAML key is services; service is not equivalent.

A failure worth recognizing

Change the db dependency in a temporary copy of the file from db to database and run docker compose config --quiet. Compose should reject a reference to a service that does not exist. Restore the correct key afterward. This checks the declarative model without damaging containers or persistent data.

> **CREDENTIAL VISIBILITY** docker compose config can print interpolated values and passwords in environment entries. Do not paste its output into public issue trackers without reviewing and redacting sensitive data. Chapter 19 replaces this demonstrative password pattern with safer secret handling.

11.11 Interpret Project Names and Generated Resources

Compose isolates the resources of different applications using a project name. Our top-level name: docklab11 makes the names predictable even if you place the folder somewhere else. You may override the project name for controlled experiments with -p; if you do, you create a separate set of networks, containers, and named volumes.

INSPECT THE COMPOSE PROJECT AND OWNED RESOURCES

```
docker compose ls
docker compose ps -a
docker network ls
docker volume ls
```

Look for resources with the docklab11 prefix and Compose labels. A name printed by docker network ls or docker volume ls is evidence; it is safer than assuming the engine will use the exact same naming scheme on every Compose release.

FORCE THE INTENDED PROJECT NAME FOR A VALIDATION-ONLY OPERATION

```
docker compose -p docklab11 config --quiet
```

Do not run two live variants of this example on the same host port 8088. A distinct Compose project gets new resources, but it still competes for the same host port. Chapter 12 teaches environment-specific overrides and optional services.

11.12 Checkpoint: Rebuild the Stack from a Clean Directory

You are now the operator. Without referring to an external repository, recreate the four canonical files, validate the Compose model, build the web service, launch both services, and collect the evidence below. Do not treat merely seeing “running” as proof the system is working.

Proof	Observable acceptance condition
Configuration	docker compose config --quiet exits successfully.
Image	docker image ls docklab11-web lists tag 1.0.
Service health	docker compose ps shows db healthy and web running.
Networking	Name db resolves from inside web; no db host port appears.
HTTP operation	http://127.0.0.1:8088 returns reachable and HTTP 200.
Failure behavior	Stopping db makes web return HTTP 503.
Persistent state	SQL row remains after down followed by up -d.
Clean exit	docker compose down removes stack, retains named volume.

Recovering when the checkpoint fails

Symptom	First evidence to collect	Likely repair
8088 is unavailable	docker compose ps; web logs	Free the host port or choose a free localhost port.
web cannot resolve db	docker compose config; service networks	Check DB_HOST=db and shared data network.
db never becomes healthy	docker compose logs db; inspect volume	Check database init logs, disk, and mount path.
Old table missing	docker volume ls; compose config	Find wrong volume/project or an accidental data deletion.
web returns 503 after db restarts	pg_isready, then web request	Wait for db readiness and check TCP reachability.

Safe finish

STOP AND REMOVE LAB CONTAINERS/NETWORKS, RETAIN TEST RECORDS

```
docker compose down
docker volume ls
```

Keep the volume if you want to revisit your exercise. If you choose to delete all lab database data, first confirm its exact volume name and that no important information is stored there. The destructive option docker compose down --volumes is intentionally not part of the default cleanup sequence.

You have verified health-gated startup, name-based networking, persistence, and a known failure mode. Chapter 12 adds environment overrides; Chapter 14 connects real DeployBoard data to PostgreSQL.

Chapter 12 — Create Repeatable Development Environments

One application. A predictable workspace. No guessing which configuration won.

Yesterday the web service started. Today it binds to a different port, a teammate sees another greeting, and an optional tool is running when nobody asked for it. None of these is a Docker failure; each is a configuration decision that has become invisible. This chapter makes those decisions explicit and repeatable.

> **YOUR DELIVERY** A development stack you can launch with one Compose command, inspect, change deliberately, and stop without discarding database records. The web process is the small demonstration service from Chapter 11; the full DeployBoard API arrives in Part IV.

Prerequisites: Docker Engine or Docker Desktop, Compose v2, Buildx, and a text editor. Use an empty folder named docklab12. Commands are one line unless labeled otherwise and work in PowerShell, Bash, and zsh. Browser checks use http://127.0.0.1:8090; if that port is occupied, change WEB_PORT in .env. Example values are for an isolated local lab, not a public deployment.

12.1 Compare Environment Injection with Compose Interpolation

Compose performs two separate jobs. Interpolation fills placeholders in the Compose model before a container is created. Injection puts environment variables inside the running service. The .env file is typically an input to the first job; it does not automatically do the second job. You must reference a variable under environment or supply it through env_file for the container to receive it.

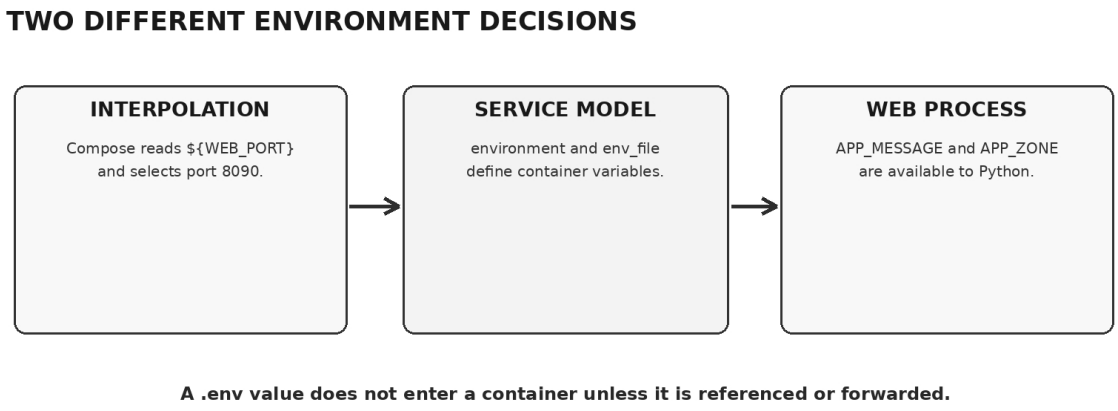


Figure 12.1 — Compose interpolation and container environment injection are separate stages.

An .env fragment used for interpolation

```
WEB_PORT=8090
APP_MESSAGE=hello-from-dot-env
```

Compose fragment that consumes these values

```
ports:
  - "127.0.0.1:${WEB_PORT:-8090}:8000"
environment:
  APP_MESSAGE: "${APP_MESSAGE:-hello-from-compose}"
```

The first substitution selects a host port, while the second creates APP_MESSAGE in the container. By contrast, APP_ZONE will arrive via env_file in the next section. Keep these channels separate when troubleshooting: the rendered service model and the process environment answer different questions.

12.2 Use .env, env_file, and environment Deliberately

Build a compact Compose demonstration as a new, standalone project. It retains Chapter 11's two-service design: a Python HTTP server checks whether PostgreSQL is reachable through the private data network. The additions in this chapter expose a mode, a message, a build marker, and a zone so that configuration changes are observable.

Create this exact project tree (write all files below)

```
docklab12/
  .env
  .gitignore
  web.runtime.env
  compose.yaml
  compose.dev.yaml
  compose.watch.yaml
  web/
    app.py
    Dockerfile
    .dockerignore
```

File: web/app.py — complete Python source

```
"""DockLab 12: disposable HTTP demonstration, not full DeployBoard."""
import json
import os
import socket
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

BUILD_MARKER = "source-v1"  # Change to source-v2 for the Watch lab.
DB_HOST = os.getenv("DB_HOST", "db")
DB_PORT = int(os.getenv("DB_PORT", "5432"))


class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path not in ("/", "/health"):
            self.send_error(404, "Use / or /health")
            return
        try:
            with socket.create_connection((DB_HOST, DB_PORT), timeout=2):
                database = "reachable"
        except OSError:
            database = "unreachable"
        payload = {
            "service": "docklab12-web",
            "build_marker": BUILD_MARKER,
            "mode": os.getenv("APP_MODE", "unset"),
            "message": os.getenv("APP_MESSAGE", "unset"),
            "zone": os.getenv("APP_ZONE", "unset"),
            "database_tcp": database,
        }
        body = json.dumps(payload, indent=2).encode("utf-8")
        self.send_response(200 if database == "reachable" else 503)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)


if __name__ == "__main__":
    server = ThreadingHTTPServer(("0.0.0.0", 8000), Handler)
    print("docklab12 listening on 8000", flush=True)
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.server_close()
```

File: web/Dockerfile

```
FROM python:3.13-slim
WORKDIR /app
COPY app.py /app/app.py
USER 65532:65532
EXPOSE 8000
CMD ["python", "-u", "/app/app.py"]
```

File: web/.dockerignore

```
__pycache__/
*.pyc
.env
.git/
```

File: .env — local interpolation inputs

```
# Development-only demonstration; never reuse this password publicly.
WEB_PORT=8090
APP_MESSAGE=hello-from-dot-env
APP_MODE=standard
DB_PASSWORD=local-demo-only-change-me
```

File: web.runtime.env — container environment inputs

```
# Values inserted into the web container by env_file.
APP_MESSAGE=hello-from-runtime-file
APP_MODE=runtime-file
APP_ZONE=training-zone
```

The same APP_MESSAGE key appears in both files on purpose. That will allow you to prove which value actually reaches the process. The database password is a conspicuously disposable classroom value; .env and docker compose config are not suitable secret stores. Chapter 19 covers real credential handling.

File: compose.yaml — complete base stack (part 1 of 2)

```
name: docklab12

services:
  web:
    build:
      context: ./web
    image: docklab12-web:1.0
    env_file:
      - ./web.runtime.env
    environment:
      DB_HOST: db
      DB_PORT: "5432"
      APP_MESSAGE: "${APP_MESSAGE:-hello-from-compose}"
      APP_MODE: "${APP_MODE:-standard}"
    ports:
      - "127.0.0.1:${WEB_PORT:-8090}:8000"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8000/health',
timeout=2).read()"]
      interval: 10s
      timeout: 3s
      retries: 3
      start_period: 15s
    restart: on-failure:3
    networks: [edge, data]
```

File: compose.yaml — continued immediately (part 2 of 2)

```
  db:
    image: postgres:17
    environment:
      POSTGRES_USER: demo
      POSTGRES_PASSWORD: "${DB_PASSWORD:?Set DB_PASSWORD in .env}"
      POSTGRES_DB: demo
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U demo -d demo"]
      interval: 4s
      timeout: 3s
      retries: 15
      start_period: 10s
    restart: unless-stopped
    networks: [data]

  toolbox:
    image: busybox:1.36
    profiles: [tools]
    command: ["sh", "-c", "nslookup db"]
    networks: [data]

networks:
  edge: {}
  data:
    internal: true

volumes:
  db_data: {}
```

Write both listings into one compose.yaml, in the printed order. The web service publishes only a localhost port; PostgreSQL has no host port. The db_data named volume holds PostgreSQL 17 files at its documented /var/lib/postgresql/data path. A healthy database starts the web service, but the web process must still handle later failures.

Validate, render, and launch from inside docklab12

```
docker compose config --quiet
docker compose config --environment
docker compose up -d --build
docker compose ps
```

Expected: Compose accepts the configuration and creates docklab12 services and networks. After startup, db and web should become healthy. Use a browser at http://127.0.0.1:8090, or the following PowerShell/Bash-compatible command:

```
docker compose exec -T web python -c "import urllib.request;
print(urllib.request.urlopen('http://127.0.0.1:8000/health').read().decode())"
```

The JSON should include message "hello-from-dot-env", zone "training-zone", and database_tcp "reachable". Values may appear in a different order. The actual successful HTTP code is 200. Do not paste real passwords into screenshots of docker compose config.

12.3 Understand Variable Precedence and Safe Defaults

There are two distinct precedence questions. For interpolation, an exported shell variable normally overrides the .env value. For a container's final environment, explicit environment entries override env_file entries; a one-off docker compose run -e can override both for that run. A Dockerfile ENV would be below these Compose-specified values.

Source / situation	Observed APP_MESSAGE
web.runtime.env alone	hello-from-runtime-file
compose.yaml environment + .env	hello-from-dot-env
Shell APP_MESSAGE=from-shell	from-shell (on a new run/recreate)
docker compose run -e APP_MESSAGE=one-off	one-off (only in that run)

Check the current interpolation and container values

```
docker compose config --environment
docker compose run --rm --no-deps web python -c "import os; print(os.getenv('APP_MESSAGE'));
print(os.getenv('APP_ZONE'))"
```

Expected: hello-from-dot-env followed by training-zone. To prove the one-off override without modifying your main web container, run:

```
docker compose run --rm --no-deps -e APP_MESSAGE=one-off web python -c "import os;
print(os.getenv('APP_MESSAGE'))"
```

Expected: one-off. --no-deps prevents starting dependencies for this inspection command; --rm removes its temporary container. An existing web container will not change its environment until it is recreated. For a shell-override experiment, use the appropriate platform form:

Bash / zsh only

```
APP_MESSAGE=from-shell docker compose config --environment
```

PowerShell only

```
$env:APP_MESSAGE = "from-shell"
docker compose config --environment
Remove-Item Env:APP_MESSAGE
```

For missing values use ${NAME:-fallback}; for required values use ${NAME:?error message}. Our DB_PASSWORD entry uses the second pattern so a missing database setting cannot pass silently. The - form accepts an explicitly empty value; the :- form also replaces empty values. Keep important credentials out of examples meant for production.

12.4 Create Development Overrides and Additional Compose Files

Rather than editing the base stack every time you work locally, use an overlay file. Compose merges the files from left to right; the later file can replace a mapping value or add a mount. In multi-file setups, resolve relative paths against the base Compose file. Inspect the merged model before assuming an override has applied.

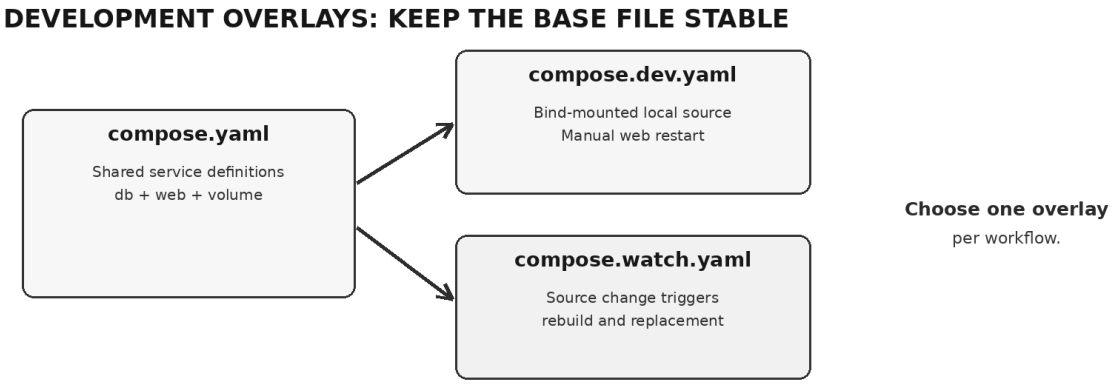


Figure 12.2 — Two alternative development overlays extend the same base stack.

File: compose.dev.yaml — complete development overlay

```
# Development overlay: source is visible immediately, but Python
# is not an auto-reloader; restart web after editing app.py.
services:
  web:
    environment:
      APP_MODE: development
    volumes:
      - type: bind
        source: ./web
        target: /app
        read_only: true
```

One-command development launch (second line)

```
docker compose -f compose.yaml -f compose.dev.yaml config --quiet
docker compose -f compose.yaml -f compose.dev.yaml up -d --build
```

Now check the web JSON again. The mode is "development". The read-only bind mount maps your host web folder to /app. A Python source-file edit is now visible inside the container, but this HTTP server is not a development auto-reloader: restart the web process to use changed Python code.

Apply a source edit with a controlled web restart

```
docker compose -f compose.yaml -f compose.dev.yaml restart web
```

DECISION Use the base file alone for a clean packaged image. Use compose.dev.yaml when you want a live host directory and can explicitly restart the process. Use the Watch overlay in Section 12.6 as a separate workflow; do not combine the bind mount with the Watch rebuild demonstration.

12.5 Use Profiles for Optional Services

The toolbox service is declared with profiles: [tools], so it does not start during the default up. This is useful for optional diagnostic tools, migrations, and one-off tasks that should not always consume resources.

Activate the optional toolbox

```
docker compose ps
docker compose config --profiles
docker compose --profile tools up -d toolbox
docker compose ps
docker compose stop toolbox
```

The profile is selected through --profile, while the service's name remains toolbox. A targeted docker compose run toolbox also activates that targeted profiled service; it does not automatically start every tool in the profile. This chapter's toolbox only resolves the database hostname on the private network; it does not read database data.

12.6 Use Bind Mounts and Compose Watch Where Supported

A bind mount is simple and transparent, but it reflects host filesystem performance and permission behavior. Compose Watch instead watches selected paths and can rebuild or sync a running service. Watch requires Compose 2.22.0 or later; check docker compose version before using it. Our demonstration chooses action: rebuild to avoid pretending that this minimal Python server hot-reloads automatically.

File: compose.watch.yaml — complete Watch overlay

```
# Alternative to the bind-mount overlay; do not combine both.
# Requires Docker Compose v2.22.0 or newer.
services:
  web:
    environment:
      APP_MODE: watch
    develop:
      watch:
        - action: rebuild
          path: ./web
```

Switch from the bind-mount workflow to Watch (foreground)

```
docker compose -f compose.yaml -f compose.dev.yaml down
docker compose -f compose.yaml -f compose.watch.yaml up --watch
```

While Watch is running, edit web/app.py: change BUILD_MARKER from "source-v1" to "source-v2", then save. Compose should rebuild the web image and replace the web service. When it is healthy, the JSON should report build_marker "source-v2" and mode "watch". Press Ctrl+C to stop the attached Watch session, then restore the marker to source-v1 if you want the original checkpoint output.

Watch also supports sync and sync+restart actions when the target filesystem and application process are configured for them. They are not interchangeable: syncing files does not necessarily make a non-reloading process read new code. For this classroom service, rebuild is the honest, reliable option.

12.7 Run One-Off Tasks with docker compose run

Use run for a disposable command that needs the same image and configuration as a declared service. It is appropriate for debugging, preflight checks, and future migrations; it is not a replacement for the long-running web process. By default, docker compose run does not publish the service's ports.

Run and remove a temporary web container

```
docker compose up -d --build
docker compose run --rm --no-deps web python -c "import platform; print(platform.python_version())"
docker compose run --rm --no-deps -e APP_MESSAGE=inspection web python -c "import os;
print(os.environ['APP_MESSAGE'])"
```

The first one-off displays the Python version from the image. The second displays inspection without changing the already running web service's APP_MESSAGE. The --no-deps option is intentional here; remove it for tasks that actually require an available database. Avoid one-off writes to a production database until you understand migrations and backups.

12.8 Manage Service Logs, Rebuilds, and Recreates

Use status and evidence before rebuilding. Logs show process output; config shows the resolved model; inspect shows the running container state.

A rebuild changes an image, whereas a restart reuses an existing container configuration. Recreate applies changed service settings.

Goal	Command / effect
Inspect recent web log output	docker compose logs --tail 25 web
Rebuild & recreate web	docker compose up -d --build --force-recreate web
Restart same container	docker compose restart web
Check active services	docker compose ps
Inspect merged configuration	docker compose config

A controlled web service refresh

```
docker compose logs --tail 25 web
docker compose up -d --build --force-recreate web
docker compose ps
```

Do not treat a successful exit code from up as proof that every user-facing operation works. Verify /health and the database connection afterward. In a later chapter, DeployBoard will have real API-level smoke tests; here you can already observe status and reachability.

12.9 Add Health Checks and Explain the Limits of Restart Policies

Both services define health checks in compose.yaml. PostgreSQL uses pg_isready; the web service calls its own /health endpoint. The web endpoint reflects database TCP reachability, not a complete SQL correctness proof. Service dependencies with condition: service_healthy affect the initial startup sequence; they do not continuously re-orchestrate a failing stack.

Simulate a dependency failure after a successful start

```
docker compose ps
docker compose stop db
docker compose ps
docker compose logs --tail 15 web
```

With db stopped, the web process remains running but a request to /health returns HTTP 503. Its health status can become unhealthy after the configured retries. restart: on-failure:3 only reacts to a process exit with a nonzero status; it is not an automatic restart whenever a health check fails. Recovery requires action and a working dependency.

Recover the database and verify the web path

```
docker compose start db
docker compose ps
docker compose exec -T web python -c "import urllib.request;
print(urllib.request.urlopen('http://127.0.0.1:8000/health').status)"
```

If the health check runs before PostgreSQL is fully ready, wait for a healthy db and repeat the web request. For real production operation you would also add application retries, logging, monitored alerts, and an incident procedure. Never confuse a restart policy with a high-availability system.

12.10 Avoid Accidental Volume Deletion During Cleanup

Named volumes have a different lifetime from containers. Docker Compose down normally removes project containers and networks while retaining named volumes. By contrast, down --volumes requests removal of declared named volumes and anonymous volumes attached to containers. Do not substitute the latter as an innocent cleanup shortcut.

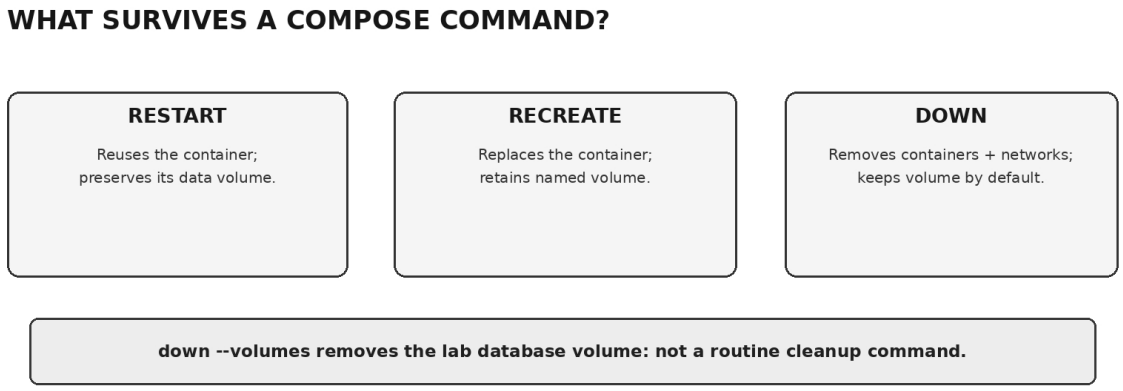


Figure 12.3 — Routine Compose commands do not all have the same data-loss consequences.

Record a value, stop the project, and prove persistence

```
docker compose exec -T db psql -U demo -d demo -c "CREATE TABLE IF NOT EXISTS workshop_notes(note text);"
docker compose exec -T db psql -U demo -d demo -c "INSERT INTO workshop_notes VALUES ('kept-after-down');"
docker compose down
docker compose up -d --build
docker compose exec -T db psql -U demo -d demo -Atc "SELECT note FROM workshop_notes;"
```

Expected query result: kept-after-down. You can run the exercise more than once, in which case the row appears more than once. This is safe classroom test data, but still never run down --volumes on a project whose data you intend to keep. File: .gitignore is included to avoid accidentally checking the local .env file into version control:

```
.env
*.env.local
__pycache__/
*.pyc
```

SAFE CLEANUP After finishing, use docker compose down. The docklab12_db_data volume remains available for later reuse. If you deliberately want to destroy the lab database after separately verifying backups, you may remove that specific lab volume; never prune volumes globally as routine cleanup.

12.11 Checkpoint: Launch One-Command Development

A dependable development environment is one you can reproduce from a clean folder, explain without guessing, and stop without destroying its records. Complete this short release rehearsal.

Acceptance test	Evidence of success
Configuration is valid	docker compose -f compose.yaml -f compose.dev.yaml config --quiet exits 0
One-command start works	docker compose -f compose.yaml -f compose.dev.yaml up -d --build
Web and database are healthy	docker compose ps shows running/healthy services
Settings are predictable	JSON mode=development, message=hello-from-dot-env, zone=training-zone
One-off override is isolated	run -e shows one-off, steady web output unchanged
Persistent records remain	A query after down/up returns kept-after-down
Optional service remains optional	toolbox absent from normal up; available via tools profile
Safe shutdown is demonstrated	docker compose down leaves docklab12_db_data intact

If a checkpoint fails, start with docker compose config and docker compose logs rather than deleting containers or volumes blindly. Your next task is not another command to memorize; it is to decide which piece of evidence will tell you what changed.

WHAT YOU CAN NOW OPERATE A Compose development environment with predictable configuration, optional services, multiple local workflows, reproducible startup, observable health, and safe persistence. In Chapter 13, DeployBoard evolves beyond this demonstration HTTP service into a real application API.

PART IV

ENGINEER A COMPLETE CONTAINERIZED APPLICATION

Chapter 13 — Expand DeployBoard into a Real API with Contracts and Authentication

THE ENGINEERING MISSION Turn the starter DeployBoard service into a small but complete API whose public behavior, identity rules, failure responses, and persistent records are testable before the PostgreSQL integration in Chapter 14.

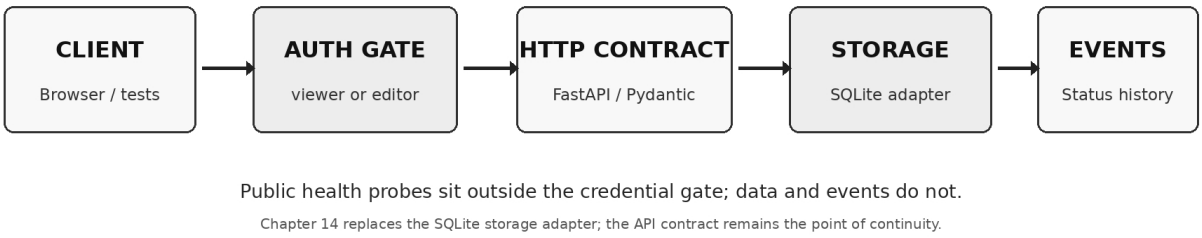


Figure 13.1 One request crosses authorization, validation, persistence, and event history.

A container that launches is not yet an application that can be trusted. The difference is observable behavior: clear contracts, durable enough local state, explicit permissions, predictable failure modes, and repeatable tests. This chapter moves DeployBoard across that boundary without pretending that a local practice stack is a public production service.

The exercise tracks deployment records across development, staging, and production environments. Readers will create a deployment, advance its status, inspect its event history, and archive it without silently erasing the audit trail. That provides a specific reason to use every API method and error code instead of building decorative CRUD endpoints.

By the end, you can prove…	Required evidence
The HTTP contract is intentional	201 / 200 / 204 / 401 / 403 / 404 / 409 / 422 tests
The state changes persist	SQLite records survive an API restart; deletion archives
Two roles have different powers	Viewer reads; editor reads and changes
The deployment boundary is explicit	Loopback-only Compose port; no claim of Internet readiness

Reproducibility contract: every application file, test, secret generator, Dockerfile, and Compose definition needed for this chapter appears below. The bundled ZIP is only a convenience. The edition uses the Chapter 13 deployment/event workload; the later Appendix E capstone expands and refactors DeployBoard into its PostgreSQL/Redis task-and-worker workload. Treat that later kit as an intentional evolution, not a byte-for-byte copy of this intermediate stage.

13.1 Specify DeployBoard's Real-World Use Cases

Begin with behaviors that an operator could recognize. A release record is a promise about what ran, where it ran, and what happened next. The chapter API does not deploy software; it records deployment outcomes in a controlled, inspectable way. This boundary makes the exercise safe to run on a laptop.

Actor	User story	Acceptance signal
Release editor	Create a deployment record for a named environment and version	POST returns 201; record begins as queued
Release editor	Record that deployment started, succeeded, or failed	PATCH enforces allowed state moves; an event is appended
Reviewer	Inspect history without modifying it	GET events returns an ordered timeline
Observer	List records only for staging or prod	GET uses validated environment filter
Operator	Archive a record but retain evidence	DELETE returns 204; normal queries hide the record
Health monitor	Check process separately from storage	/healthz and /readyz have different meanings

Contract decision No browser login, password reset, user database, OAuth provider, real CI trigger, remote rollout, or rate limiter is hidden behind these endpoints. Those are separate engineering tasks. A two-token lab model is adequate to prove authorization logic, not to protect a public multi-tenant API.

13.2 Define Deployments, Environments, and Status Events

An environment is the destination class: dev, staging, or prod. A deployment record names the environment, the immutable proposed version string, current state, and timestamps. A status event is append-only evidence that something happened to that record. Keeping these concepts separate allows the API to return a concise deployment while preserving a history of transitions.

Object	Fields / allowed values	Rule
Environment	dev, staging, prod	Known identifiers only; no user-created environments in v1
Deployment	id, environment, version, status, created_at, updated_at	ID/timestamps generated server-side
Status event	id, deployment_id, kind, detail, occurred_at	Created and updated inside the same DB transaction
Archive marker	archived_at	Hides record from normal reads; does not delete event rows

From	Allowed next states	Illegal transitions
queued	running or failed	queued → succeeded
running	succeeded or failed	running → queued
succeeded	none	Any later change
failed	none	Any later change

The first migration is intentionally tiny. SQLite supports a single-file local adapter without asking a beginner to configure a database server. Chapter 14 replaces this learning adapter with PostgreSQL migrations and pool-aware connections. Do not copy the SQLite file onto a production server as though it were an operational backup design.

13.3 Create CRUD Routes and Request Models

Use a small contract module to make client-owned inputs distinguishable from server-owned outputs. The create model accepts exactly environment and version; the change model accepts only status. Unknown fields fail rather than being ignored. The response contract includes generated metadata but never a credential or internal database path.

FILE app/schemas.py / 31 lines / exact reproducible listing

```
"""Version 1 HTTP contracts; clients must not send server-owned fields."""
from datetime import datetime
from typing import Literal
from pydantic import BaseModel, ConfigDict, Field

EnvironmentSlug = Literal["dev", "staging", "prod"]
```

```
DeploymentState = Literal["queued", "running", "succeeded", "failed"]

class DeploymentCreate(BaseModel):
    model_config = ConfigDict(extra="forbid")
    environment: EnvironmentSlug
    version: str = Field(pattern=r"^[A-Za-z0-9][A-Za-z0-9._-]{0,63}$")

class DeploymentChange(BaseModel):
    model_config = ConfigDict(extra="forbid")
    status: Literal["running", "succeeded", "failed"]

class DeploymentOut(BaseModel):
    id: int
    environment: EnvironmentSlug
    version: str
    status: DeploymentState
    created_at: str
    updated_at: str

class EventOut(BaseModel):
    id: int
    deployment_id: int
    kind: str
    detail: str
    occurred_at: str
```

Below is the complete persistence adapter. It initializes the learning schema, uses parameterized SQL, groups the deployment and its event in one transaction, and serializes writes with BEGIN IMMEDIATE. The database module returns ordinary dictionaries; the HTTP module decides what status codes to send.

FILE app/storage.py / 174 lines / exact reproducible listing

```
"""SQLite learning adapter; replaced by PostgreSQL work in Chapter 14."""
import os
import sqlite3
from contextlib import contextmanager
from datetime import datetime, timezone
from pathlib import Path

def now_utc() -> str:
    return datetime.now(timezone.utc).isoformat(timespec="seconds")

def db_path() -> Path:
    return Path(os.getenv("DB_PATH", "./data/deployboard.sqlite"))

def connect() -> sqlite3.Connection:
    con = sqlite3.connect(db_path(), timeout=5, isolation_level=None)
    con.row_factory = sqlite3.Row
    con.execute("PRAGMA foreign_keys = ON")
    con.execute("PRAGMA busy_timeout = 5000")
    return con

@contextmanager
def database():
    con = connect()
    try:
        yield con
    finally:
        con.close()

def initialize() -> None:
    db_path().parent.mkdir(parents=True, exist_ok=True)
```

```
    with database() as con:
        con.executescript("""
        CREATE TABLE IF NOT EXISTS deployments (
          id INTEGER PRIMARY KEY,
          environment TEXT NOT NULL CHECK(environment IN ('dev','staging','prod')),
          version TEXT NOT NULL,
          status TEXT NOT NULL DEFAULT 'queued',
          created_at TEXT NOT NULL,
          updated_at TEXT NOT NULL,
          archived_at TEXT
        );
        CREATE TABLE IF NOT EXISTS status_events (
          id INTEGER PRIMARY KEY,
          deployment_id INTEGER NOT NULL REFERENCES deployments(id),
          kind TEXT NOT NULL,
          detail TEXT NOT NULL,
          occurred_at TEXT NOT NULL
        );
        CREATE INDEX IF NOT EXISTS ix_deployments_env ON deployments(environment);
        CREATE INDEX IF NOT EXISTS ix_events_deployment ON status_events(deployment_id);
        """)

def visible(con: sqlite3.Connection, deployment_id: int):
    return con.execute(
        "SELECT * FROM deployments WHERE id = ? AND archived_at IS NULL",
        (deployment_id,),
    ).fetchone()

def payload(row) -> dict:
    return {key: row[key] for key in (
        "id", "environment", "version", "status", "created_at", "updated_at")}

def create(environment: str, version: str) -> dict:
    stamp = now_utc()
    with database() as con:
        con.execute("BEGIN IMMEDIATE")
        try:
            cur = con.execute(
                "INSERT INTO deployments(environment, version, status, created_at, updated_at) "
                "VALUES (?, ?, 'queued', ?, ?)", (environment, version, stamp, stamp)
            )
            identifier = cur.lastrowid
            con.execute(
                "INSERT INTO status_events(deployment_id, kind, detail, occurred_at) "
                "VALUES (?, 'created', 'queued', ?)", (identifier, stamp)
            )
            result = payload(visible(con, identifier))
            con.execute("COMMIT")
            return result
        except Exception:
            con.execute("ROLLBACK")
            raise

def list_deployments(environment: str | None = None) -> list[dict]:
    with database() as con:
        if environment is None:
            rows = con.execute(
                "SELECT * FROM deployments WHERE archived_at IS NULL ORDER BY id"
            ).fetchall()
        else:
            rows = con.execute(
                "SELECT * FROM deployments WHERE archived_at IS NULL "
                "AND environment = ? ORDER BY id", (environment,)
            ).fetchall()
        return [payload(row) for row in rows]
```

```
def get(deployment_id: int) -> dict | None:
    with database() as con:
        row = visible(con, deployment_id)
        return payload(row) if row else None

def events(deployment_id: int) -> list[dict] | None:
    with database() as con:
        if not visible(con, deployment_id):
            return None
        rows = con.execute(
            "SELECT id, deployment_id, kind, detail, occurred_at "
            "FROM status_events WHERE deployment_id = ? ORDER BY id",
            (deployment_id,),
        ).fetchall()
        return [dict(row) for row in rows]

ALLOWED = {"queued": {"running", "failed"},
           "running": {"succeeded", "failed"},
           "succeeded": set(), "failed": set()}

def change(deployment_id: int, new_status: str) -> tuple[str, dict | None]:
    with database() as con:
        con.execute("BEGIN IMMEDIATE")
        try:
            row = visible(con, deployment_id)
            if not row:
                con.execute("ROLLBACK")
                return "missing", None
            if new_status not in ALLOWED[row["status"]]:
                con.execute("ROLLBACK")
                return "conflict", None
            stamp = now_utc()
            con.execute("UPDATE deployments SET status = ?, updated_at = ? WHERE id = ?",
                        (new_status, stamp, deployment_id))
            con.execute(
                "INSERT INTO status_events(deployment_id, kind, detail, occurred_at) "
                "VALUES (?, 'status_changed', ?, ?)", (deployment_id, new_status, stamp)
            )
            result = payload(visible(con, deployment_id))
            con.execute("COMMIT")
            return "ok", result
        except Exception:
            con.execute("ROLLBACK")
            raise

def archive(deployment_id: int) -> bool:
    with database() as con:
        con.execute("BEGIN IMMEDIATE")
        try:
            if not visible(con, deployment_id):
                con.execute("ROLLBACK")
                return False
            stamp = now_utc()
            con.execute("UPDATE deployments SET archived_at = ?, updated_at = ? "
                        "WHERE id = ?", (stamp, stamp, deployment_id))
            con.execute(
                "INSERT INTO status_events(deployment_id, kind, detail, occurred_at) "
                "VALUES (?, 'archived', 'soft_delete', ?)", (deployment_id, stamp)
            )
            con.execute("COMMIT")
            return True
        except Exception:
            con.execute("ROLLBACK")
            raise
```

13.4 Validate Input and Return Predictable Errors

A usable API is legible when callers make mistakes. FastAPI/Pydantic validates strings, enumerations, and unexpected fields before application code runs. The service gives a deliberate 409 for an impossible state transition, while a missing record produces 404. A malformed token is different from a valid viewer attempting an editor action.

Status	Meaning	Example proof
201	Record created	POST valid deployment with editor token
200	Read or accepted state change	GET record; PATCH queued → running
204	Record archived without response body	DELETE valid deployment
401	No acceptable bearer authentication	No token or Basic scheme
403	Credentials authenticated but insufficient or wrong	Viewer attempts POST; random bearer
404	Resource absent or archived	GET id 9999
409	State transition forbidden	queued → succeeded
422	Request shape/value incorrect	Extra id field or invalid environment
503	Readiness dependency unavailable	Database read failure

CONTRACT EXAMPLE / body accepted by POST /v1/deployments

```
{
  "environment": "staging",
  "version": "1.2.3"
}
```

CONTRACT EXAMPLE / expected created-response shape

```
{
  "id": 1,
  "environment": "staging",
  "version": "1.2.3",
  "status": "queued",
  "created_at": "<UTC timestamp>",
  "updated_at": "<UTC timestamp>"
}
```

The timestamp placeholder above illustrates structure; the server supplies real UTC times. Invalid requests return FastAPI's standard detail body. Do not assert exact framework-generated wording in the tests: match the HTTP code and the public fields whose semantics you control.

13.5 Build the Health and Readiness Endpoints

Liveness means the HTTP process can respond. Readiness means its essential data dependency is usable right now. /healthz returns a small static response; /readyz queries the SQLite database. Neither endpoint returns configuration values or a token. A passing probe is not a substitute for authenticated CRUD or data-integrity tests.

VERIFY / host-bound Compose listener

```
curl -i http://127.0.0.1:8088/healthz
# HTTP/1.1 200 ... {"status":"alive"}
curl -i http://127.0.0.1:8088/readyz
# HTTP/1.1 200 ... {"status":"ready"}
```

If /healthz is 200 but /readyz is 503, do not restart blindly: examine the database path, volume permissions, and service log first. These endpoints intentionally answer different questions. The Compose health check uses /readyz, not /healthz.

13.6 Structure the Python Application for Container Deployment

The HTTP layer should be thin enough to understand in one reading. auth.py decides who may act, schemas.py defines allowable input/output, storage.py maintains local state and events, and main.py maps operations to routes. FastAPI lifespan initializes the small practice database once per application start.

FILE app/main.py / 83 lines / exact reproducible listing

```
"""DeployBoard Chapter 13: authenticated CRUD and an inspectable status timeline."""
import sqlite3
from contextlib import asynccontextmanager
from fastapi import Depends, FastAPI, HTTPException, Query, Response
from app.auth import can_read, can_write
from app.schemas import (DeploymentChange, DeploymentCreate, DeploymentOut,
                         EnvironmentSlug, EventOut)
from app import storage

@asynccontextmanager
async def lifespan(app: FastAPI):
    storage.initialize()
    yield

app = FastAPI(title="DeployBoard API", version="1.0.0", lifespan=lifespan)

@app.get("/healthz")
def healthz():
    return {"status": "alive"}
```

```
@app.get("/readyz")
def readyz():
    try:
        with storage.database() as con:
            con.execute("SELECT 1 FROM deployments LIMIT 1").fetchall()
    except (sqlite3.Error, OSError):
        raise HTTPException(503, "database unavailable")
    return {"status": "ready"}

@app.get("/v1/environments")
def environments(_role: str = Depends(can_read)):
    return [{"slug": slug, "label": label} for slug, label in (
        ("dev", "Development"), ("staging", "Staging"), ("prod", "Production"))]

@app.post("/v1/deployments", status_code=201, response_model=DeploymentOut)
def create_deployment(data: DeploymentCreate, _role: str = Depends(can_write)):
    return storage.create(data.environment, data.version)

@app.get("/v1/deployments", response_model=list[DeploymentOut])
def list_deployments(environment: EnvironmentSlug | None = Query(default=None),
                     _role: str = Depends(can_read)):
    return storage.list_deployments(environment)

@app.get("/v1/deployments/{deployment_id}", response_model=DeploymentOut)
def get_deployment(deployment_id: int, _role: str = Depends(can_read)):
    result = storage.get(deployment_id)
    if result is None:
        raise HTTPException(404, "deployment not found")
    return result

@app.patch("/v1/deployments/{deployment_id}", response_model=DeploymentOut)
def change_deployment(deployment_id: int, data: DeploymentChange,
                      _role: str = Depends(can_write)):
    code, result = storage.change(deployment_id, data.status)
    if code == "missing":
        raise HTTPException(404, "deployment not found")
    if code == "conflict":
        raise HTTPException(409, "invalid status transition")
    return result

@app.get("/v1/deployments/{deployment_id}/events", response_model=list[EventOut])
def deployment_events(deployment_id: int, _role: str = Depends(can_read)):
    result = storage.events(deployment_id)
    if result is None:
        raise HTTPException(404, "deployment not found")
    return result

@app.delete("/v1/deployments/{deployment_id}", status_code=204)
def archive_deployment(deployment_id: int, _role: str = Depends(can_write)):
    if not storage.archive(deployment_id):
        raise HTTPException(404, "deployment not found")
    return Response(status_code=204)
```

FILE app/__init__.py / 1 lines / exact reproducible listing

```
"""DeployBoard Chapter 13 teaching package."""
```

PROJECT TREE / all chapter-owned inputs

```
DeployBoard_Chapter13/
  app/{__init__.py,auth.py,main.py,schemas.py,storage.py}
  scripts/generate_tokens.py
  tests/{conftest.py,test_api.py}
  Dockerfile  compose.yaml  .dockerignore  .gitignore
```

```
requirements.txt   requirements-dev.txt   README.md
secrets/     # generated locally; not committed or built
data/        # local-only; not committed or built
```

In Chapter 14, keep the public contract stable while replacing the storage adapter and adding PostgreSQL migrations. When the eventual capstone brings back the task/report-worker model, reconcile endpoint resources explicitly instead of copying two competing main.py files over each other.

13.7 Add Configuration for Development and Testing

Store the learning database path and secret file paths outside the source. The API reads DB_PATH, TOKEN_READ_FILE, and TOKEN_WRITE_FILE. Unit tests substitute a fresh temporary database and distinct 48-character tokens, so a test cannot mutate the learner's real data. Production secrets, TLS certificates, and identity policies require stronger controls later.

FILE requirements.txt / 2 lines / exact reproducible listing

```
fastapi==0.128.2
uvicorn==0.48.0
```

FILE requirements-dev.txt / 3 lines / exact reproducible listing

```
-r requirements.txt
httpx==0.28.1
pytest==9.0.2
```

FILE scripts/generate_tokens.py / 20 lines / exact reproducible listing

```
"""Create untracked local token files; do not paste token contents into docs."""
from pathlib import Path
from secrets import token_urlsafe
import os

folder = Path("secrets")
folder.mkdir(mode=0o700, exist_ok=True)
folder.chmod(0o700)
for name in ("read_token.txt", "write_token.txt"):
    path = folder / name
    if path.exists():
        print(f"Kept existing {path}")
        continue
    # The parent directory is private (0700); a readable file is needed
    # for a non-root UID inside some file-backed Compose secret mounts.
    fd = os.open(path, os.O_CREAT | os.O_EXCL | os.O_WRONLY, 0o644)
    with os.fdopen(fd, "w", encoding="utf-8") as stream:
        stream.write(token_urlsafe(48) + "\n")
    path.chmod(0o644)
    print(f"Created {path}; keep the parent directory private")
```

FILE .gitignore / 7 lines / exact reproducible listing

```
.venv/
.pytest_cache/
```

```
__pycache__/
*.py[cod]
*.sqlite*
data/
secrets/
```

Local secret permissions On POSIX systems the generator keeps secrets/ traversable only by its owner (0700). The files are 0644 *inside that private directory* to support non-root access with Compose file-backed secret mounts, which cannot reliably remap source-file UID/GID. This is a local teaching compromise, not production secret hygiene. Windows users must check ACLs separately. Do not add another container to the same trust boundary; replace this mechanism for public deployments.

OPTION A / create and test locally in a virtual environment

```
python -m venv .venv
# Activate .venv using your operating system's shell.
python -m pip install -r requirements-dev.txt
python scripts/generate_tokens.py
python -m pytest -q
```

Run from the project root. A bare local uvicorn invocation requires setting all three environment variables first; the Compose file supplies the container paths automatically. For Python directly, set DB_PATH to a writable absolute filename and TOKEN_*_FILE to your generated absolute paths. If a credential file is missing, an authenticated request fails rather than accepting a default password; check secret paths before launch.

13.8 Write Functional API and Input-Validation Tests

The quality gate must prove permissions *and* behavior. These tests create a new SQLite database under pytest's tmp_path fixture for each case. TestClient runs application startup through its context manager, which exercises schema initialization. The suite checks authorization, CRUD, history, forbidden transitions, unexpected fields, and test isolation.

FILE tests/conftest.py / 26 lines / exact reproducible listing

```
import pytest
from fastapi.testclient import TestClient
from app.main import app

@pytest.fixture
def api(tmp_path, monkeypatch):
    reader = tmp_path / "read.txt"
    editor = tmp_path / "write.txt"
    reader.write_text("r" * 48, encoding="utf-8")
    editor.write_text("e" * 48, encoding="utf-8")
```

```
    monkeypatch.setenv("DB_PATH", str(tmp_path / "deployboard.sqlite"))
    monkeypatch.setenv("TOKEN_READ_FILE", str(reader))
    monkeypatch.setenv("TOKEN_WRITE_FILE", str(editor))
    with TestClient(app) as client:
        yield client

@pytest.fixture
def read_headers():
    return {"Authorization": "Bearer " + "r" * 48}

@pytest.fixture
def write_headers():
    return {"Authorization": "Bearer " + "e" * 48}
```

FILE tests/test_api.py / 115 lines / exact reproducible listing

```
def make(api, write_headers, version="1.2.3", env="staging"):
    return api.post("/v1/deployments", headers=write_headers,
                    json={"environment": env, "version": version})

def test_health_and_readiness_are_distinct(api):
    assert api.get("/healthz").json() == {"status": "alive"}
    assert api.get("/readyz").json() == {"status": "ready"}

def test_requires_bearer_token(api, read_headers, write_headers):
    assert api.get("/v1/deployments").status_code == 401
    assert api.get("/v1/deployments", headers={"Authorization": "Basic x"}).status_code == 401
    assert api.get("/v1/deployments", headers={"Authorization": "Bearer wrong"}).status_code == 403
    assert api.get("/v1/deployments", headers=read_headers).status_code == 200
    assert api.get("/v1/environments", headers=write_headers).status_code == 200

def test_viewer_cannot_modify(api, read_headers):
    result = api.post("/v1/deployments", headers=read_headers,
                      json={"environment": "dev", "version": "1.0"})
    assert result.status_code == 403

def test_validation_errors_and_no_extra_fields(api, write_headers):
    for body in ({"environment": "production", "version": "1.0"},
                 {"environment": "dev", "version": "bad version"},
                 {"environment": "dev", "version": "1.0", "id": 99}):
        assert api.post("/v1/deployments", headers=write_headers, json=body).status_code == 422

def test_crud_audit_and_filter(api, write_headers, read_headers):
    created = make(api, write_headers)
    assert created.status_code == 201
    record = created.json()
    identifier = record["id"]
    assert record["status"] == "queued"
    assert len(api.get("/v1/deployments", headers=read_headers).json()) == 1
    assert api.get("/v1/deployments?environment=dev", headers=read_headers).json() == []
    assert api.get(f"/v1/deployments/{identifier}", headers=read_headers).status_code == 200
    running = api.patch(f"/v1/deployments/{identifier}", headers=write_headers,
                        json={"status": "running"})
    assert running.status_code == 200
    assert running.json()["status"] == "running"
    done = api.patch(f"/v1/deployments/{identifier}", headers=write_headers,
                     json={"status": "succeeded"})
    assert done.status_code == 200
    assert api.patch(f"/v1/deployments/{identifier}", headers=write_headers,
                     json={"status": "failed"}).status_code == 409
    events = api.get(f"/v1/deployments/{identifier}/events", headers=read_headers).json()
    assert [row["kind"] for row in events] == ["created", "status_changed", "status_changed"]
    assert api.delete(f"/v1/deployments/{identifier}", headers=write_headers).status_code == 204
    assert api.get(f"/v1/deployments/{identifier}", headers=read_headers).status_code == 404
    assert api.get("/v1/deployments", headers=read_headers).json() == []
```

```
def test_conflict_and_missing_resources(api, write_headers, read_headers):
    identifier = make(api, write_headers).json()["id"]
    assert api.patch(f"/v1/deployments/{identifier}", headers=write_headers,
                     json={"status": "succeeded"}).status_code == 409
    assert api.delete("/v1/deployments/9999", headers=write_headers).status_code == 404
    assert api.get("/v1/deployments/9999/events", headers=read_headers).status_code == 404

def test_reader_cannot_patch_or_delete(api, write_headers, read_headers):
    identifier = make(api, write_headers).json()["id"]
    assert api.patch(f"/v1/deployments/{identifier}", headers=read_headers,
                     json={"status": "running"}).status_code == 403
    assert api.delete(f"/v1/deployments/{identifier}", headers=read_headers).status_code == 403

def test_strict_patch_contract(api, write_headers):
    identifier = make(api, write_headers).json()["id"]
    assert api.patch(f"/v1/deployments/{identifier}", headers=write_headers,
                     json={"status": "running", "environment": "prod"}).status_code == 422

def test_isolated_test_database(api, read_headers):
    assert api.get("/v1/deployments", headers=read_headers).json() == []

def test_invalid_environment_filter(api, read_headers):
    assert api.get("/v1/deployments?environment=qa", headers=read_headers).status_code == 422

def test_readiness_fails_if_database_cannot_open(api, tmp_path, monkeypatch):
    monkeypatch.setenv("DB_PATH", str(tmp_path / "missing" / "db.sqlite"))
    assert api.get("/healthz").status_code == 200
    result = api.get("/readyz")
    assert result.status_code == 503
    assert result.json()["detail"] == "database unavailable"

def test_reinitialize_preserves_existing_records(api, read_headers, write_headers):
    from app import storage
    identifier = make(api, write_headers).json()["id"]
    storage.initialize()  # Simulates harmless schema initialization after restart.
    result = api.get(f"/v1/deployments/{identifier}", headers=read_headers)
    assert result.status_code == 200
    assert result.json()["version"] == "1.2.3"

def test_archived_record_retains_status_events(api, write_headers):
    from app import storage
    identifier = make(api, write_headers).json()["id"]
    assert api.delete(f"/v1/deployments/{identifier}", headers=write_headers).status_code == 204
    with storage.database() as con:
        audit = con.execute(
            "SELECT kind FROM status_events WHERE deployment_id = ? ORDER BY id", (identifier,)
        ).fetchall()
        record = con.execute(
            "SELECT archived_at FROM deployments WHERE id = ?", (identifier,)
        ).fetchone()
    assert [row["kind"] for row in audit] == ["created", "archived"]
    assert record["archived_at"] is not None
```

OFFLINE TEST RECEIPT / measured in the publishing environment

```
python -m pytest -q
.............  [100%]
13 passed in 0.32s
```

The publishing-environment test receipt above was produced with Python 3.13.5, FastAPI 0.128.2, HTTPX 0.28.1, and pytest 9.0.2. The container Dockerfile targets Python 3.12-slim; a live Docker build and platform-

specific runs were not available in this environment. Record your own outcomes rather than transferring the offline green check to an untested host.

13.9 Build the DeployBoard Service Image

A deliberate Dockerfile is readable: select a maintained base image, install only runtime packages, copy the application, and run as a non-root user. This exercise declares port 8000 as metadata and does not make it publicly reachable until Compose publishes a host port. The image tag is a teaching reference, not an immutable supply-chain guarantee: resolve and record a digest before a real release.

FILE Dockerfile / 12 lines / exact reproducible listing

```
FROM python:3.12-slim
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
WORKDIR /srv
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt \
    && groupadd -g 10001 app \
    && useradd -u 10001 -g app -M -s /usr/sbin/nologin app \
    && mkdir -p /data && chown app:app /data
COPY --chown=app:app app ./app
USER 10001:10001
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

FILE .dockerignore / 11 lines / exact reproducible listing

```
.git
.venv
.pytest_cache
__pycache__
*.py[cod]
*.sqlite
*.sqlite-*
data/
secrets/
tests/
README.md
```

IMAGE CHECKS / execute on a Docker-equipped host

```
docker build --pull -t deployboard:chapter13 .
docker image inspect deployboard:chapter13 \
  --format '{{.Config.User}} | {{.Config.ExposedPorts}}'
# Expect a non-root UID/GID and 8000/tcp metadata.
```

The application uses a writable /data directory for the SQLite learning file and a non-root process. Named-volume ownership can vary by container host; confirm the mounted directory permits UID 10001 to write. If it does

not, investigate the volume's ownership rather than running an application as root by habit. Leave volume deletion out of troubleshooting commands.

13.10 Use Compose to Launch the API Reliably

Compose creates a single service for this chapter. The published port binds to 127.0.0.1 to keep the teaching API local to the Docker host; the database is internal to a named volume. Two file-backed secrets mount inside the container. Restart policy improves resilience to a stopped process but does not solve invalid configuration or database corruption.

FILE compose.yaml / 29 lines / exact reproducible listing

```
services:
  api:
    build: .
    ports:
      - "127.0.0.1:8088:8000"
    environment:
      DB_PATH: /data/deployboard.sqlite
      TOKEN_READ_FILE: /run/secrets/read_token
      TOKEN_WRITE_FILE: /run/secrets/write_token
    secrets:
      - read_token
      - write_token
    volumes:
      - db_data:/data
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8000/readyz', timeout=2)"]
      interval: 15s
      timeout: 3s
      retries: 3
      start_period: 10s

secrets:
  read_token:
    file: ./secrets/read_token.txt
  write_token:
    file: ./secrets/write_token.txt
volumes:
  db_data:
```

START / OBSERVE / STOP / from the project root

```
python scripts/generate_tokens.py
docker compose config --quiet
docker compose up --build -d
docker compose ps
docker compose logs --tail=40 api
curl -sS http://127.0.0.1:8088/readyz
# Stop containers; keep db_data volume:
docker compose down
```

Avoid docker compose down -v for this lesson: -v deletes the named SQLite volume and the records you are meant to preserve. During verification, stop and recreate the container with docker compose down followed by docker

compose up -d; confirm a previously created record still exists. Do not assume host loopback binding, file-backed Compose secrets, or Linux container permissions behave identically on every Desktop backend. Check your actual Docker Engine and Compose versions.

13.11 Choose a Practical API Token Authentication Model

Three common paths are (1) no credential checks, suitable only for throwaway stubs, (2) fixed role tokens, easy to test but difficult to rotate at scale, and (3) provider-backed identity with expiry, scopes, revocation, and audit. Choose fixed tokens only for this supervised local exercise. The read token may call GET routes; the write token includes read permission and may POST, PATCH, and DELETE.

FILE app/auth.py / 37 lines / exact reproducible listing

```
"""Simple, intentional two-role local token model (not an identity provider)."""
import hmac
import os
from pathlib import Path
from fastapi import Header, HTTPException

def _token_from_file(variable: str) -> str:
    name = os.getenv(variable)
    if not name:
        raise RuntimeError(f"Configure {variable} with a secret file path")
    token = Path(name).read_text(encoding="utf-8").strip()
    if len(token) < 32:
        raise RuntimeError(f"Secret file for {variable} is too short")
    return token

def _role(authorization: str | None, write: bool) -> str:
    scheme, sep, value = (authorization or "").partition(" ")
    if not sep or scheme.lower() != "bearer" or not value.strip():
        raise HTTPException(401, "Bearer token required", headers={"WWW-Authenticate": "Bearer"})
    presented = value.strip()
    reader = _token_from_file("TOKEN_READ_FILE")
    editor = _token_from_file("TOKEN_WRITE_FILE")
    if hmac.compare_digest(presented, editor):
        return "editor"
    if not write and hmac.compare_digest(presented, reader):
        return "viewer"
    raise HTTPException(403, "Insufficient permissions")

def can_read(authorization: str | None = Header(default=None)) -> str:
    return _role(authorization, False)

def can_write(authorization: str | None = Header(default=None)) -> str:
    return _role(authorization, True)
```

Security boundary Both token files are required and must be at least 32 characters. The comparison uses hmac.compare_digest. A wrong bearer

token receives 403, malformed or missing authentication receives 401, and a viewer attempting a write also receives 403. This deliberately narrow model has no per-user identities, expiration, revocation API, rate limiter, or TLS termination; none of those capabilities should be implied by a passing local test.

13.12 Authenticate API Requests and Enforce Access Permissions

Endpoint / operation	Anonymous	Viewer token	Editor token
GET /healthz or /readyz	200 when healthy	200	200
GET /v1/environments	401	200	200
GET /v1/deployments and /{id}/events	401	200 / 404	200 / 404
POST /v1/deployments	401	403	201 / 422
PATCH /v1/deployments/{id}	401	403	200 / 404 / 409 / 422
DELETE /v1/deployments/{id}	401	403	204 / 404

For a shell demonstration below, execute only against 127.0.0.1. The local shell variable avoids putting tokens into the manuscript, but a command-line authorization header can still be visible to local process inspection. Do not paste tokens into tickets, screenshots, recorded terminals, or public command history.

BASH / POSIX EXERCISE / run after Compose is healthy

```
WRITE_TOKEN="$(cat secrets/write_token.txt)"
READ_TOKEN="$(cat secrets/read_token.txt)"
curl -sS -X POST http://127.0.0.1:8088/v1/deployments \
  -H "Authorization: Bearer ${WRITE_TOKEN}" \
  -H 'Content-Type: application/json' \
  -d '{"environment":"staging","version":"1.2.3"}'
curl -sS http://127.0.0.1:8088/v1/deployments \
  -H "Authorization: Bearer ${READ_TOKEN}"
# Use the actual returned id in PATCH and GET /events.
```

UPDATE / AUDIT / ARCHIVE / replace id 1 with your actual returned id

```
curl -sS -i -X PATCH http://127.0.0.1:8088/v1/deployments/1 \
  -H "Authorization: Bearer ${WRITE_TOKEN}" \
  -H 'Content-Type: application/json' -d '{"status":"running"}'
curl -sS http://127.0.0.1:8088/v1/deployments/1/events \
  -H "Authorization: Bearer ${READ_TOKEN}"
curl -sS -i -X DELETE http://127.0.0.1:8088/v1/deployments/1 \
  -H "Authorization: Bearer ${WRITE_TOKEN}"
# Expect 204; then a GET of id 1 returns 404.
```

POWERSHELL ALTERNATIVE / reader-only request

```
$readToken = (Get-Content .\secrets\read_token.txt -Raw).Trim()
$headers = @{ Authorization = "Bearer $readToken" }
Invoke-RestMethod -Uri http://127.0.0.1:8088/v1/deployments `
  -Headers $headers -Method Get
```

An editor token is not interchangeable with the viewer token for write tests. After creating a record, try the same POST using the viewer token and record HTTP 403. Then try an impossible state change with the editor token and record 409. This is stronger evidence than a screenshot of the Swagger UI.

13.13 Checkpoint: Verify Real CRUD Behavior and Protected Endpoints

Treat the checkpoint as a release receipt. A feature is not complete just because code exists; identify the command, the actual output, and the failure response that demonstrates its boundary. You should be able to reproduce the evidence after rebuilding the image and recreating the container.

Gate	Pass condition	Evidence to retain
BUILD	Image builds; non-root identity inspected	Build logs and image inspection
VERIFY 1	All 13 offline tests pass	pytest output + Python/package versions
VERIFY 2	201 create, 200 read, 200 update, 204 archive	Redacted responses with ids and timestamps
BREAK 1	Anonymous GET denied; viewer POST denied	401 and 403 proof
BREAK 2	Invalid body and impossible state rejected	422 and 409 proof
REPAIR	Real cause diagnosed; tests re-run	Change note and evidence
SHIP (local)	Loopback port, Compose healthy, data survives recreate	compose config, ps, readyz and persistence check

RELEASE RECEIPT / copy into your operator notes

Chapter 13 revision / Git commit: _____
Date, OS, CPU architecture: _____
Python / FastAPI / Compose / Engine versions: _____
Test run and result (actual): _____
Build image identity / digest, if available: _____
Access checks 401 / 403: _____
CRUD statuses 201 / 200 / 204 / 404: _____
Validation 422 and conflict 409: _____
Persistence after recreate: _____
Known limitations / outstanding risks: _____
Decision: LOCAL PRACTICE PASS / HOLD Reviewer: _____

BREAK / REPAIR LAB Intentionally change a request field to an unknown name and watch 422; then repair the payload. Attempt queued → succeeded and watch 409; first move queued → running. Remove the read header and watch 401; restore it without weakening the endpoint. For a controlled readiness drill, temporarily point DB_PATH to a non-writable location in a disposable test session—not in a valued data volume—and verify 503 or startup failure. Record the observation rather than making the check pass artificially.

What moves forward: the resource contracts, validation rules, role boundaries, and acceptance tests. What changes in Chapter 14: SQLite gives way to PostgreSQL, schema migrations become explicit, secrets and connection-pool limits are revisited, and restore evidence becomes necessary. The later multi-service chapters add Redis, workers, and proxy boundaries. This is the difference between “my container starts” and “I know what the release is allowed to do.”

Chapter 14 — Add PostgreSQL, Migrations, and Persistent Records

A container can be replaced in seconds. A lost deployment history cannot. In this chapter you move DeployBoard from Chapter 13's in-memory repository to a relational PostgreSQL database without changing the HTTP contracts or viewer/operator access rules that already work.

The outcome is specific: a schema with two related tables, a versioned migration, a private PostgreSQL service, a bounded connection pool, idempotent sample data, and real records that remain after the API and database containers are recreated.

BEFORE / CHAPTER 13	AFTER / CHAPTER 14
Records disappear with the API process	Records live in a named PostgreSQL volume
Python memory enforces transitions	Database transactions preserve transitions
No migration or schema revision	Alembic tracks an explicit schema revision
Readiness checks only local configuration	Readiness checks credentials and a queryable table

WORKING BOUNDARY *This is a single-host development deployment, not a fully hardened public database. The Compose .env file is a local learning convenience. Production secrets, separate database roles, backups, and operations are expanded in later chapters.*

14.1 Design a Minimal Relational Data Model

DeployBoard needs only two tables: deployments stores the current release state, and status_events records how a release reached that state. Each event belongs to one deployment; deleting a deployment deletes its events through a foreign-key cascade. An event sequence is unique within a deployment so that readers can inspect a stable history.

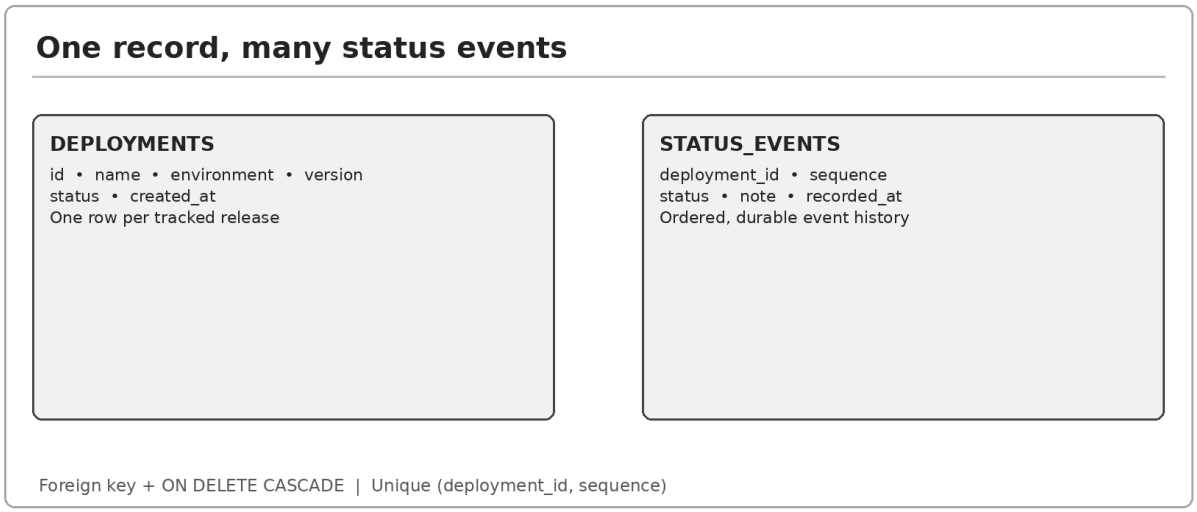


Figure 14.1 Deployments and their durable event history.

Do not add the background-job tables planned for Chapter 15. This chapter is about turning the Chapter 13 API into a persistent service with the smallest useful schema.

This SQLAlchemy Core metadata is the schema definition used for queries and migration comparisons. Alembic will perform the actual PostgreSQL table creation.

FILE / app/db.py

```
"""Chapter 14 relational schema. Alembic, not create_all(), owns production DDL."""
from sqlalchemy import (
    Column, DateTime, ForeignKey, Integer, MetaData, String,
    Table, UniqueConstraint, func,
)

metadata = MetaData()

deployments = Table(
    "deployments", metadata,
    Column("id", Integer, primary_key=True),
    Column("name", String(60), nullable=False),
    Column("environment", String(20), nullable=False),
    Column("version", String(40), nullable=False),
    Column("status", String(20), nullable=False),
    Column("created_at", DateTime(timezone=True),
           server_default=func.now(), nullable=False),
)

status_events = Table(
    "status_events", metadata,
```

```
    Column("id", Integer, primary_key=True),
    Column("deployment_id", Integer,
           ForeignKey("deployments.id", ondelete="CASCADE"),
           nullable=False),
    Column("sequence", Integer, nullable=False),
    Column("status", String(20), nullable=False),
    Column("note", String(200), nullable=False),
    Column("recorded_at", DateTime(timezone=True),
           server_default=func.now(), nullable=False),
    UniqueConstraint("deployment_id", "sequence", name="uq_event_sequence"),
)
```

14.2 Add PostgreSQL to the Compose Stack

Give the application and database separate services on a private, user-defined bridge network. The API is reachable only from the local machine on port 8000; the database has no ports mapping. A third, opt-in service runs migrations as a discrete release operation.

Create the following complete Compose file. The shared YAML environment anchor keeps application and migration credentials consistent without duplicating every setting.

FILE / compose.yaml

```
name: deployboard-ch14
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_USER: deployboard
      POSTGRES_DB: deployboard
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in
.env}
    volumes:
      - dbdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
      interval: 5s
      timeout: 3s
      retries: 10
      start_period: 15s
    restart: unless-stopped
    networks: [private]

  api:
    build: .
```

```
    image: deployboard:ch14
    ports: ["127.0.0.1:8000:8000"]
    environment: &app_env
      DB_HOST: db
      DB_PORT: "5432"
      DB_NAME: deployboard
      DB_USER: deployboard
      DB_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in .env}
      DEPLOYBOARD_VIEWER_TOKEN: ${DEPLOYBOARD_VIEWER_TOKEN:?set viewer
token}
      DEPLOYBOARD_OPERATOR_TOKEN: ${DEPLOYBOARD_OPERATOR_TOKEN:?set
operator token}
    depends_on:
      db: {condition: service_healthy}
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request;
urllib.request.urlopen('http://127.0.0.1:8000/ready')"]
      interval: 10s
      timeout: 3s
      retries: 3
      start_period: 15s
    restart: unless-stopped
    networks: [private]

  migrate:
    image: deployboard:ch14
    profiles: [tools]
    environment: *app_env
    depends_on:
      db: {condition: service_healthy}
    command: ["alembic", "upgrade", "head"]
    networks: [private]

volumes:
  dbdata:
networks:
  private:
    driver: bridge
```

The database image uses PostgreSQL 17. For version 17 and earlier, the durable volume must be mounted at /var/lib/postgresql/data; PostgreSQL 18 and later use a different official-image volume layout. Do not change majors by editing the tag in place.

The API and migration runner use the same image, so schema files are the same ones deployed with the application.

FILE / Dockerfile

```
FROM python:3.13-slim
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1
WORKDIR /service
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
RUN groupadd --system app && \
    useradd --system --gid app --no-create-home app
COPY --chown=app:app app ./app
COPY --chown=app:app migrations ./migrations
COPY --chown=app:app alembic.ini ./alembic.ini
COPY --chown=app:app tools ./tools
USER app
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Retain the FastAPI versions introduced in Chapter 13 and add the PostgreSQL driver, SQLAlchemy, and Alembic. Pin versions for this lab; update them through a separately tested release.

FILE / requirements.txt

```
fastapi==0.128.2
uvicorn==0.48.0
pydantic==2.13.5
SQLAlchemy==2.0.44
alembic==1.17.1
psycopg[binary]==3.2.12
```

Local tests use these additional development dependencies.

FILE / requirements-dev.txt

```
-r requirements.txt
pytest==9.0.2
httpx==0.28.1
```

14.3 Configure the Database Without Publishing Its Port

PostgreSQL must be reachable from the API through the Compose service name db, not through localhost and not through a fixed container IP. In the Compose file there is intentionally no db.ports key. Inspect this when troubleshooting: the absence of a published port is a deliberate security boundary.

Copy this template to .env in the same folder as compose.yaml. Replace all REPLACE_ values with distinct, strong lab credentials; never commit .env. The viewer and operator tokens must each contain at least 20 characters and must differ.

FILE / .env.example

```
# Copy to .env and replace ALL placeholders before running.
POSTGRES_PASSWORD=REPLACE_WITH_LOCAL_ONLY_DB_PASSWORD
DEPLOYBOARD_VIEWER_TOKEN=REPLACE_WITH_RANDOM_VIEWER_TOKEN
DEPLOYBOARD_OPERATOR_TOKEN=REPLACE_WITH_RANDOM_OPERATOR_TOKEN
```

These ignore rules reduce the chance of accidentally committing credentials and local test artifacts.

FILE / .gitignore

```
.env
*.pyc
__pycache__/
.pytest_cache/
```

The build context must not include local credentials, test databases, or unrelated version-control material.

FILE / .dockerignore

```
.git
__pycache__/
*.pyc
.pytest_cache/
.env
.env.*
!.env.example
tests/
*.zip
*.db
```

Create your local environment file

```
# Bash / zsh
cp .env.example .env
# PowerShell
Copy-Item .env.example .env
```

Open .env in a local editor and replace the placeholders. PostgreSQL initialization values apply only when its data directory is empty; changing .env later does not change an existing database password. A secure random token can be generated with python -c "import secrets; print(secrets.token_urlsafe(32))". Generate separate tokens for viewer and

operator. Prefer a strong alphanumeric lab database password here; URL.create in the application also handles reserved characters safely.

> **CHECK BEFORE SHARING** *docker compose config expands environment values and may print credentials. Use it locally for validation; do not paste the full output into public issues, screenshots, or log archives.*

14.4 Create Connection Settings and Pool Boundaries

A web request should borrow a database connection for a short operation, then return it. The API uses SQLAlchemy's QueuePool with pool_size=3 and max_overflow=2: one API process can open at most five checked-out connections. These are small demonstration limits, not universal production recommendations.

Keep the two-token model intact and add explicit PostgreSQL connection fields. The API fails closed if credentials are missing or left as placeholders.

FILE / app/settings.py

```
"""Environment-driven configuration; do not embed credentials in images."""
from dataclasses import dataclass
import os
import secrets


@dataclass(frozen=True)
class Settings:
    viewer_token: str = ""
    operator_token: str = ""
    service_label: str = "DeployBoard"
    db_host: str = "db"
    db_port: int = 5432
    db_name: str = "deployboard"
    db_user: str = "deployboard"
    db_password: str = ""

    @classmethod
    def from_env(cls) -> "Settings":
        return cls(
            viewer_token=os.getenv("DEPLOYBOARD_VIEWER_TOKEN", ""),
            operator_token=os.getenv("DEPLOYBOARD_OPERATOR_TOKEN", ""),
            service_label=os.getenv("DEPLOYBOARD_LABEL", "DeployBoard"),
            db_host=os.getenv("DB_HOST", "db"),
            db_port=int(os.getenv("DB_PORT", "5432")),
            db_name=os.getenv("DB_NAME", "deployboard"),
```

```
        db_user=os.getenv("DB_USER", "deployboard"),
        db_password=os.getenv("DB_PASSWORD", ""),
    )

@property
def valid_tokens(self) -> bool:
    return (
        len(self.viewer_token) >= 20
        and len(self.operator_token) >= 20
        and not self.viewer_token.startswith("REPLACE_")
        and not self.operator_token.startswith("REPLACE_")
        and not secrets.compare_digest(
            self.viewer_token, self.operator_token
        )
    )

@property
def valid_database(self) -> bool:
    return bool(self.db_host and self.db_name and
                self.db_user and self.db_password and
                not self.db_password.startswith("REPLACE_"))
```

The URL.create helper in app/main.py assembles the database URL without treating punctuation in a password as URL syntax. pool_pre_ping handles many stale idle connections during checkout, but does not replay an operation that failed halfway through a transaction.

SETTING	WHY IT MATTERS
pool_size = 3	Keeps up to three persistent pooled connections per API process
max_overflow = 2	Allows two additional temporary connections during a spike
pool_timeout = 10	Bounds the wait for a free connection
pool_pre_ping = True	Tests and refreshes stale connections on checkout

14.5 Integrate the Application with PostgreSQL

Instead of rewriting the HTTP routes, preserve Chapter 13’s input models, error responses, authentication dependencies, and status transition rules. Replace MemoryStore with SQLStore. Each mutation uses a transaction; event insertion and current-status changes commit together.

This repository implements create, list, get, patch, append-event, and delete. For PostgreSQL, SELECT ... FOR UPDATE serializes concurrent status

changes to the same deployment.

FILE / app/store.py

```
"""Database-backed repository preserving Chapter 13's API contract."""
from sqlalchemy import delete, func, insert, select, text, update
from sqlalchemy.engine import Engine
from sqlalchemy.exc import SQLAlchemyError

from .db import deployments, status_events
from .models import (Deployment, DeploymentCreate, DeploymentPatch,
                     Environment, EventCreate, Status, StatusEvent)


class MissingDeployment(Exception):
    pass


class InvalidTransition(Exception):
    pass


ALLOWED = {
    Status.pending: {Status.running, Status.failed},
    Status.running: {Status.succeeded, Status.failed},
    Status.failed: {Status.running},
    Status.succeeded: set(),
}


class SQLStore:
    def __init__(self, engine: Engine) -> None:
        self.engine = engine

    def is_ready(self) -> bool:
        try:
            with self.engine.connect() as cx:
                cx.execute(text("SELECT 1"))
                cx.execute(select(deployments.c.id).limit(1))
            return True
        except SQLAlchemyError:
            return False

    def _get(self, cx, item_id: int) -> Deployment:
        row = cx.execute(
            select(deployments).where(deployments.c.id == item_id)
        ).mappings().first()
        if row is None:
            raise MissingDeployment()
```

```
    event_rows = cx.execute(
        select(status_events)
        .where(status_events.c.deployment_id == item_id)
        .order_by(status_events.c.sequence)
    ).mappings().all()
    return Deployment(
        id=row["id"], name=row["name"],
        environment=row["environment"], version=row["version"],
        status=row["status"], created_at=row["created_at"],
        events=[StatusEvent(
            sequence=e["sequence"], status=e["status"],
            note=e["note"], recorded_at=e["recorded_at"]
        ) for e in event_rows],
    )

def get(self, item_id: int) -> Deployment:
    with self.engine.connect() as cx:
        return self._get(cx, item_id)

def list(self, environment: Environment | None) -> list[Deployment]:
    query = select(deployments.c.id).order_by(deployments.c.id)
    if environment is not None:
        query = query.where(deployments.c.environment == environment.value)
    with self.engine.connect() as cx:
        ids = cx.execute(query).scalars().all()
        return [self._get(cx, item_id) for item_id in ids]

def create(self, payload: DeploymentCreate) -> Deployment:
    with self.engine.begin() as cx:
        item_id = cx.execute(insert(deployments).values(
            name=payload.name,
            environment=payload.environment.value,
            version=payload.version,
            status=Status.pending.value,
        ).returning(deployments.c.id)).scalar_one()
    return self.get(item_id)

def update(self, item_id: int, patch: DeploymentPatch) -> Deployment:
    changes = patch.model_dump(exclude_unset=True, exclude_none=True)
    with self.engine.begin() as cx:
        present = cx.execute(
            select(deployments.c.id)
            .where(deployments.c.id == item_id).with_for_update()
        ).first()
        if present is None:
            raise MissingDeployment()
        if changes:
            cx.execute(update(deployments).where(
```

```
                deployments.c.id == item_id).values(**changes))
        return self.get(item_id)

    def append_event(self, item_id: int,
                     event: EventCreate) -> Deployment:
        with self.engine.begin() as cx:
            current = cx.execute(
                select(deployments.c.status)
                .where(deployments.c.id == item_id).with_for_update()
            ).scalar_one_or_none()
            if current is None:
                raise MissingDeployment()
            if event.status not in ALLOWED[Status(current)]:
                raise InvalidTransition()
            sequence = cx.execute(
                select(func.coalesce(func.max(status_events.c.sequence), 0))
                .where(status_events.c.deployment_id == item_id)
            ).scalar_one() + 1
            cx.execute(insert(status_events).values(
                deployment_id=item_id, sequence=sequence,
                status=event.status.value, note=event.note,
            ))
            cx.execute(update(deployments)
                       .where(deployments.c.id == item_id)
                       .values(status=event.status.value))
        return self.get(item_id)

    def delete(self, item_id: int) -> None:
        with self.engine.begin() as cx:
            result = cx.execute(delete(deployments).where(
                deployments.c.id == item_id))
            if result.rowcount == 0:
                raise MissingDeployment()
```

CONCURRENCY NOTE *SQLite's test fixture does not enforce PostgreSQL row locks. The locking behavior must be verified through PostgreSQL integration tests before using concurrent writers in production.*

The API's URL paths and access checks remain unchanged. The /health endpoint shows that the process is alive; /ready additionally verifies credentials and the database schema.

FILE / app/main.py

```
"""The Chapter 13 HTTP contract, now backed by a relational store."""
from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException, Query, Request, Response
from sqlalchemy import create_engine
```

```
from sqlalchemy.engine import URL
from sqlalchemy.exc import SQLAlchemyError

from .models import (Deployment, DeploymentCreate, DeploymentPatch,
                     Environment, EventCreate)
from .security import require_operator, require_viewer
from .settings import Settings
from .store import InvalidTransition, MissingDeployment, SQLStore


def make_engine(settings: Settings):
    url = URL.create(
        "postgresql+psycopg", username=settings.db_user,
        password=settings.db_password, host=settings.db_host,
        port=settings.db_port, database=settings.db_name,
    )
    return create_engine(url, pool_size=3, max_overflow=2,
                         pool_timeout=10, pool_pre_ping=True)


def get_store(request: Request) -> SQLStore:
    store = request.app.state.store
    if store is None:
        raise HTTPException(503, detail="Database is not configured")
    return store


def create_app(settings: Settings | None = None,
               store: SQLStore | None = None) -> FastAPI:
    api = FastAPI(title="DeployBoard API", version="0.14.0")
    api.state.settings = settings or Settings.from_env()
    if store is not None:
        api.state.store = store
    elif api.state.settings.valid_database:
        api.state.store = SQLStore(make_engine(api.state.settings))
    else:
        api.state.store = None

    @api.get("/health")
    def health() -> dict[str, str]:
        return {"status": "ok"}

    @api.get("/ready")
    def ready(request: Request, response: Response) -> dict[str, str]:
        configured = request.app.state.settings.valid_tokens
        repo = request.app.state.store
        if not configured or repo is None or not repo.is_ready():
            response.status_code = 503
```

```
        return {"status": "unready", "storage": "postgresql"}
    return {"status": "ready", "storage": "postgresql"}

@api.get("/v1/deployments", response_model=list[Deployment])
def list_deployments(
    _: Annotated[str, Depends(require_viewer)],
    repository: Annotated[SQLStore, Depends(get_store)],
    environment: Environment | None = Query(default=None),
) -> list[Deployment]:
    return repository.list(environment)

@api.post("/v1/deployments", response_model=Deployment,
          status_code=201)
def create_deployment(
    payload: DeploymentCreate,
    _: Annotated[str, Depends(require_operator)],
    repository: Annotated[SQLStore, Depends(get_store)],
) -> Deployment:
    return repository.create(payload)

@api.get("/v1/deployments/{item_id}", response_model=Deployment)
def get_deployment(
    item_id: int,
    _: Annotated[str, Depends(require_viewer)],
    repository: Annotated[SQLStore, Depends(get_store)],
) -> Deployment:
    try:
        return repository.get(item_id)
    except MissingDeployment:
        raise HTTPException(404, detail="Deployment not found") from None

@api.patch("/v1/deployments/{item_id}", response_model=Deployment)
def patch_deployment(
    item_id: int, payload: DeploymentPatch,
    _: Annotated[str, Depends(require_operator)],
    repository: Annotated[SQLStore, Depends(get_store)],
) -> Deployment:
    try:
        return repository.update(item_id, payload)
    except MissingDeployment:
        raise HTTPException(404, detail="Deployment not found") from None

@api.post("/v1/deployments/{item_id}/events",
          response_model=Deployment, status_code=201)
def add_status_event(
    item_id: int, payload: EventCreate,
    _: Annotated[str, Depends(require_operator)],
    repository: Annotated[SQLStore, Depends(get_store)],
```

```
    ) -> Deployment:
        try:
            return repository.append_event(item_id, payload)
        except MissingDeployment:
            raise HTTPException(404, detail="Deployment not found") from None
        except InvalidTransition:
            raise HTTPException(409, detail="Status transition not allowed") from None

    @api.delete("/v1/deployments/{item_id}", status_code=204)
    def delete_deployment(
        item_id: int,
        _: Annotated[str, Depends(require_operator)],
        repository: Annotated[SQLStore, Depends(get_store)],
    ) -> Response:
        try:
            repository.delete(item_id)
        except MissingDeployment:
            raise HTTPException(404, detail="Deployment not found") from None
        return Response(status_code=204)

    return api


app = create_app()
```

Two short Chapter 13 files remain exactly unchanged. They are repeated here to make the final project reconstructible from this chapter alone rather than relying on a download or external repository.

FILE / app/models.py

```
"""Validated request and response contracts."""
from datetime import datetime
from enum import Enum
from typing import Annotated

from pydantic import BaseModel, ConfigDict, Field, StringConstraints


class Environment(str, Enum):
    development = "development"
    staging = "staging"
    production = "production"


class Status(str, Enum):
    pending = "pending"
    running = "running"
    succeeded = "succeeded"
```

```
    failed = "failed"


Name = Annotated[
    str, StringConstraints(strip_whitespace=True, min_length=3, max_length=60)
]
Version = Annotated[str, Field(pattern=r"^[A-Za-z0-9][A-Za-z0-9._-]{0,39}$")]


class DeploymentCreate(BaseModel):
    model_config = ConfigDict(extra="forbid")
    name: Name
    environment: Environment
    version: Version


class DeploymentPatch(BaseModel):
    model_config = ConfigDict(extra="forbid")
    name: Name | None = None
    version: Version | None = None


class EventCreate(BaseModel):
    model_config = ConfigDict(extra="forbid")
    status: Status
    note: Annotated[str, Field(max_length=200)] = ""


class StatusEvent(BaseModel):
    sequence: int
    status: Status
    note: str
    recorded_at: datetime


class Deployment(BaseModel):
    id: int
    name: str
    environment: Environment
    version: str
    status: Status
    created_at: datetime
    events: list[StatusEvent]
```

FILE / app/security.py

```
"""Two lab API tokens with separate read and write permissions."""
import secrets
from typing import Annotated
```

```
from fastapi import Depends, HTTPException, Request, status
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer

bearer = HTTPBearer(auto_error=False)


def require_viewer(
    request: Request,
    credentials: Annotated[
        HTTPAuthorizationCredentials | None, Depends(bearer)
    ],
) -> str:
    settings = request.app.state.settings
    if not settings.valid_tokens:
        raise HTTPException(status_code=503, detail="API tokens not configured")
    if credentials is None:
        raise HTTPException(
            status_code=401, detail="Bearer token required",
            headers={"WWW-Authenticate": "Bearer"},
        )
    token = credentials.credentials
    if secrets.compare_digest(token, settings.operator_token):
        return "operator"
    if secrets.compare_digest(token, settings.viewer_token):
        return "viewer"
    raise HTTPException(
        status_code=401, detail="Invalid token",
        headers={"WWW-Authenticate": "Bearer"},
    )


def require_operator(role: Annotated[str, Depends(require_viewer)]) -> str:
    if role != "operator":
        raise HTTPException(status_code=403, detail="Write access required")
    return role
```

FILE / app/__init__.py

```
"""DeployBoard Chapter 13 application package."""
```

14.6 Create and Apply Database Migrations

A migration is part of the application release. It must not be hidden inside normal API startup because multiple API instances could try to modify the schema simultaneously. Here, a disposable migrate service applies the versioned first revision before the API starts.

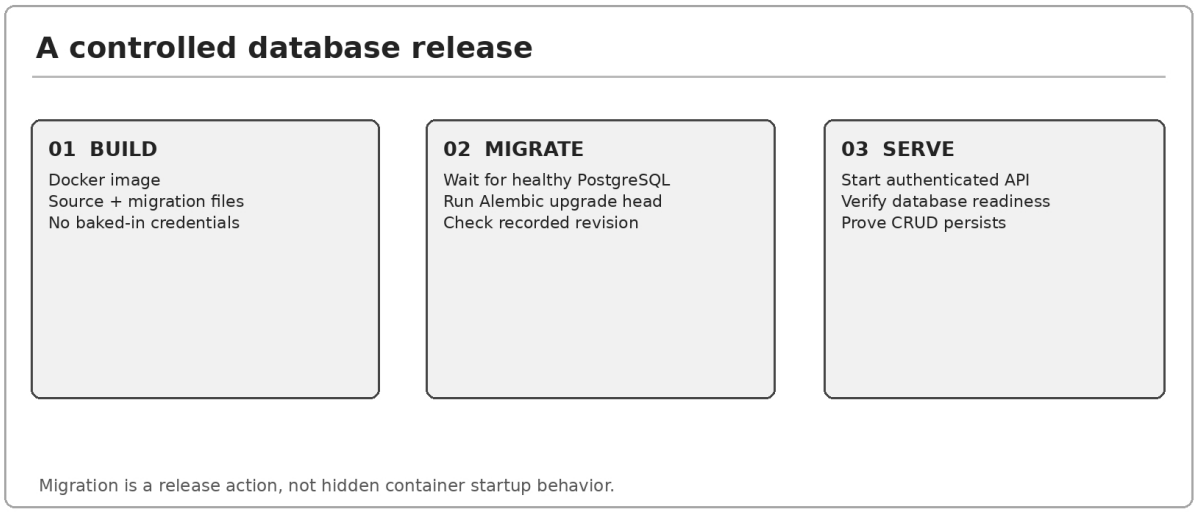


Figure 14.2 Build the image, apply the migration once, then start the application.

This minimal Alembic configuration points to the migration directory.

FILE / alembic.ini

```
[alembic]
script_location = migrations
prepend_sys_path = .
```

The migration environment connects online using the same structured credential fields as the application.

FILE / migrations/env.py

```
"""Online-only migration environment for this PostgreSQL lab."""
from alembic import context
from sqlalchemy import create_engine, pool
from sqlalchemy.engine import URL

from app.db import metadata
from app.settings import Settings


target_metadata = metadata

if context.is_offline_mode():
    raise RuntimeError("Use the online Compose migration workflow")

settings = Settings.from_env()
if not settings.valid_database:
    raise RuntimeError("Set DB_PASSWORD before running migrations")

url = URL.create(
    "postgresql+psycopg", username=settings.db_user,
```

```
    password=settings.db_password, host=settings.db_host,
    port=settings.db_port, database=settings.db_name,
)
engine = create_engine(url, poolclass=pool.NullPool)
with engine.connect() as connection:
    context.configure(connection=connection, target_metadata=target_metadata)
    with context.begin_transaction():
        context.run_migrations()
engine.dispose()
```

This checked-in revision creates the tables, uniqueness constraint, foreign key, and event lookup index.

FILE / migrations/versions/0001_initial.py

```
"""Initial DeployBoard schema.

Revision ID: 0001
Revises:
"""
from alembic import op
import sqlalchemy as sa

revision = "0001"
down_revision = None
branch_labels = None
depends_on = None


def upgrade() -> None:
    op.create_table(
        "deployments",
        sa.Column("id", sa.Integer, primary_key=True),
        sa.Column("name", sa.String(60), nullable=False),
        sa.Column("environment", sa.String(20), nullable=False),
        sa.Column("version", sa.String(40), nullable=False),
        sa.Column("status", sa.String(20), nullable=False),
        sa.Column("created_at", sa.DateTime(timezone=True),
                  server_default=sa.func.now(), nullable=False),
    )
    op.create_table(
        "status_events",
        sa.Column("id", sa.Integer, primary_key=True),
        sa.Column("deployment_id", sa.Integer,
                  sa.ForeignKey("deployments.id", ondelete="CASCADE"),
                  nullable=False),
        sa.Column("sequence", sa.Integer, nullable=False),
        sa.Column("status", sa.String(20), nullable=False),
        sa.Column("note", sa.String(200), nullable=False),
```

```
        sa.Column("recorded_at", sa.DateTime(timezone=True),
                  server_default=sa.func.now(), nullable=False),
        sa.UniqueConstraint("deployment_id", "sequence",
                            name="uq_event_sequence"),
    )
    op.create_index("ix_events_deployment", "status_events",
                    ["deployment_id"])


def downgrade() -> None:
    op.drop_index("ix_events_deployment", table_name="status_events")
    op.drop_table("status_events")
    op.drop_table("deployments")
```

Build, initialize, migrate, start

```
docker compose config --quiet
docker compose build api
docker compose up -d db
docker compose run --rm migrate
docker compose run --rm migrate alembic current
docker compose up -d api
docker compose ps
```

Expected migration evidence: Alembic reports upgrading to revision 0001, and alembic current prints 0001. A second docker compose run --rm migrate should perform no schema change. On a fresh volume, the PostgreSQL service needs time to become healthy before migrations begin.

14.7 Seed Sample Deployment Records

A seed is useful only if it is repeatable. This script checks for a named sample deployment before inserting it. It does not reset the database, delete user records, or generate a new ID on every run.

FILE / tools/seed.py

```
"""Idempotent development fixture; run from the API image."""
from app.main import make_engine
from app.models import DeploymentCreate, Environment
from app.settings import Settings
from app.store import SQLStore

repo = SQLStore(make_engine(Settings.from_env()))
existing = {item.name for item in repo.list(None)}
if "DeployBoard API" not in existing:
    item = repo.create(DeploymentCreate(
        name="DeployBoard API", environment=Environment.staging,
```

```
        version="1.0.0"))
    print(f"Seeded deployment ID {item.id}")
else:
    print("Sample deployment already exists; no duplicate inserted")
```

Run the seed twice

```
docker compose run --rm migrate python -m tools.seed
docker compose run --rm migrate python -m tools.seed
docker compose exec db psql -U deployboard -d deployboard -c "SELECT id, name, version
FROM deployments;"
```

On the first run, expect a newly seeded deployment. On the second, expect a no-duplicate message. The SQL query should show one DeployBoard API sample row even after both runs.

14.8 Use a Named Volume for Database Files

The named volume dbdata is owned by the Compose project and mapped to the PostgreSQL 17 data directory. Replacing the API image leaves records untouched. Recreating the PostgreSQL container also keeps the volume unless you explicitly remove it.

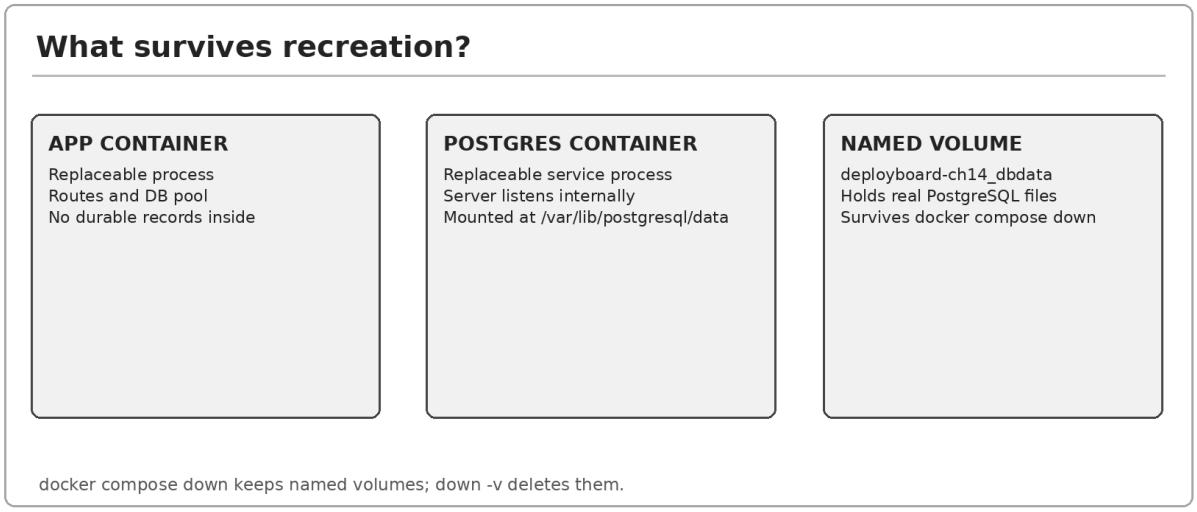


Figure 14.3 Separate replaceable containers from durable database files.

Inspect persistence resources safely

```
docker compose ps
docker volume ls --filter label=com.docker.compose.project=deployboard-ch14
docker compose exec db psql -U deployboard -d deployboard -c "SELECT COUNT(*) FROM
deployments;"
```

The resulting project volume normally has the name deployboard-ch14_dbdata. Do not edit volume contents directly, run rm -rf inside the PostgreSQL data path, or use docker compose down -v during this exercise.

14.9 Handle Database Readiness and Connection Retries

Startup ordering and ongoing availability are two different problems. Compose waits for a healthy db before starting the API, but it does not guarantee PostgreSQL stays reachable. When the database is stopped later, the API process can remain alive while /ready correctly returns HTTP 503.

This short probe prints a stable HTTP status instead of a long shell-specific one-liner.

FILE / tools/check_ready.py

```
"""Report API readiness from inside its own container."""
import urllib.error
import urllib.request

try:
    with urllib.request.urlopen(
        "http://127.0.0.1:8000/ready", timeout=5
    ) as response:
        print(f"Readiness HTTP {response.status}")
except urllib.error.HTTPError as exc:
    print(f"Readiness HTTP {exc.code}")
    raise SystemExit(1) from None
```

Observe loss and return of readiness

```
docker compose stop db
docker compose exec api python -m tools.check_ready
docker compose start db
# After db returns to healthy:
docker compose exec api python -m tools.check_ready
```

The first probe intentionally exits nonzero after printing Readiness HTTP 503. Wait for db to become healthy before the second probe; it should print Readiness HTTP 200. For a host-side check, open http://127.0.0.1:8000/ready. Neither restart policies nor pool_pre_ping make failed in-flight database transactions automatically safe to retry.

> **OPERATIONAL BOUNDARY** *This is a supervised local recovery exercise. In real applications retry only idempotent or transactionally protected operations, bound retries, and record repeated failures. Chapter 15 extends those rules to queued work.*

14.10 Verify Persistence Across Recreates and Rebuilds

The strongest evidence is a record created before containers are replaced and queried afterward. First run the smoke test, then recreate without deleting the named volume. The database record must keep the same ID and contents.

A local regression suite exercises the same HTTP contract using a disposable SQLite database fixture. PostgreSQL-specific migrations and persistence are verified with the Compose commands in this chapter.

FILE / tests/test_api.py

```
"""Fast integration-style contract tests using a disposable SQLite fixture.

PostgreSQL migration and container persistence are checked by the Compose labs.
"""
from pathlib import Path

import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine

from app.db import metadata
from app.main import create_app
from app.settings import Settings
from app.store import SQLStore

VIEWER = "test-viewer-token-of-sufficient-length"
OPERATOR = "test-operator-token-of-sufficient-length"


@pytest.fixture
def client(tmp_path: Path):
    engine = create_engine(f"sqlite+pysqlite:///{tmp_path / 'test.db'}")
    metadata.create_all(engine)  # Test fixture only; use Alembic in PostgreSQL.
    api = create_app(Settings(VIEWER, OPERATOR), SQLStore(engine))
    with TestClient(api) as test_client:
        yield test_client
    engine.dispose()


def auth(token):
    return {"Authorization": f"Bearer {token}"}
```

```
def create_item(client):
    return client.post("/v1/deployments", headers=auth(OPERATOR),
        json={"name": "DeployBoard API", "environment": "staging",
              "version": "1.0.0"})


def test_health_and_readiness(client):
    assert client.get("/health").status_code == 200
    assert client.get("/ready").json()["storage"] == "postgresql"


def test_permissions(client):
    assert client.get("/v1/deployments").status_code == 401
    assert client.post("/v1/deployments", headers=auth(VIEWER),
        json={"name": "Read only", "environment": "staging",
              "version": "1.0"}).status_code == 403


def test_crud_and_relational_event_history(client):
    created = create_item(client)
    assert created.status_code == 201
    item_id = created.json()["id"]
    changed = client.patch(f"/v1/deployments/{item_id}",
        headers=auth(OPERATOR), json={"version": "1.0.1"})
    assert changed.json()["version"] == "1.0.1"
    event = client.post(f"/v1/deployments/{item_id}/events",
        headers=auth(OPERATOR),
        json={"status": "running", "note": "Starting"})
    assert event.status_code == 201
    assert event.json()["events"][0]["sequence"] == 1
    assert client.post(f"/v1/deployments/{item_id}/events",
        headers=auth(OPERATOR), json={"status": "pending"}).status_code == 409
    assert len(client.get("/v1/deployments",
        headers=auth(VIEWER)).json()) == 1
    assert client.delete(f"/v1/deployments/{item_id}",
        headers=auth(OPERATOR)).status_code == 204
    assert client.get(f"/v1/deployments/{item_id}",
        headers=auth(VIEWER)).status_code == 404


def test_validation_and_filter(client):
    assert client.post("/v1/deployments", headers=auth(OPERATOR),
        json={"name": "x", "environment": "staging",
              "version": "1", "admin": True}).status_code == 422
    assert create_item(client).status_code == 201
    assert client.get("/v1/deployments?environment=production",
        headers=auth(VIEWER)).json() == []
```

```
def test_fail_closed_when_unconfigured():
    api = create_app(Settings())
    with TestClient(api) as client:
        assert client.get("/ready").status_code == 503
        assert client.get("/v1/deployments").status_code == 503
```

Run the local regression suite

```
python -m pip install -r requirements-dev.txt
python -m pytest -q tests/test_api.py
```

Expected: five passing tests. The Python-only tests do not prove that PostgreSQL container networking, migrations, and storage are correct; the following live stack exercises are mandatory for that claim.

Persist and recreate the running services

```
docker compose exec api python -m tools.smoke
docker compose exec db psql -U deployboard -d deployboard -c "SELECT id, name, version
FROM deployments ORDER BY id;"
docker compose down
docker compose up -d db api
docker compose exec db psql -U deployboard -d deployboard -c "SELECT id, name, version
FROM deployments ORDER BY id;"
```

The second SQL query should show the same records and IDs. docker compose down removes containers and networks but keeps named volumes by default. In contrast, docker compose down -v deletes those volumes and the database data they contain.

14.11 Checkpoint: Recover Real Application Records

You can now evaluate the system against a concrete contract: authenticated CRUD operations work, the schema is versioned, PostgreSQL is not published to the host, the API correctly reports an unready database, and a record survives a complete stack recreation.

The smoke program runs inside the API container, where its runtime token environment is already available.

FILE / tools/smoke.py

```
"""Portable HTTP smoke check (run from host with exported tokens)."""
import json
import os
import urllib.error
```

```
import urllib.request

root = os.getenv("API_URL", "http://127.0.0.1:8000")
viewer = os.environ["DEPLOYBOARD_VIEWER_TOKEN"]
operator = os.environ["DEPLOYBOARD_OPERATOR_TOKEN"]


def request(path, method="GET", token=None, data=None):
    payload = json.dumps(data).encode() if data is not None else None
    headers = {"Content-Type": "application/json"}
    if token:
        headers["Authorization"] = f"Bearer {token}"
    req = urllib.request.Request(root + path, data=payload,
                                 headers=headers, method=method)
    try:
        with urllib.request.urlopen(req, timeout=5) as r:
            return r.status, json.load(r) if r.status != 204 else None
    except urllib.error.HTTPError as e:
        return e.code, None


assert request("/ready")[0] == 200
assert request("/v1/deployments", token=viewer)[0] == 200
assert request("/v1/deployments", method="POST", token=viewer,
               data={"name": "Read only", "environment": "staging",
                     "version": "1"})[0] == 403
status, item = request("/v1/deployments", method="POST", token=operator,
                       data={"name": "Smoke release", "environment": "staging",
                             "version": "14.0"})
assert status == 201
item_id = item["id"]
assert request(f"/v1/deployments/{item_id}", token=viewer)[0] == 200
print(f"PASS: PostgreSQL-backed API, auth, and CRUD (ID {item_id})")
```

EVIDENCE	ACCEPT WHEN
Schema revision	alembic current reports 0001
Database exposure	No host port is published for db
Authentication	Viewer cannot POST; operator can
Persistence	Same record ID remains after down and up
Readiness	/ready reports 503 while db is unavailable
Volume safety	Volume remains unless deliberately removed

If a check fails, diagnose with docker compose ps, docker compose logs --tail=80 db api, docker compose run --rm migrate alembic current, and docker compose config --quiet. Do not share config output containing

credentials. A particularly common mistake is running migrations against a different Compose project or database volume.

End the lab without deleting its database

```
docker compose down
# Optional ONLY when you have intentionally finished and backed up data:
# docker compose down -v
```

You have moved DeployBoard across a meaningful boundary: the application is now durable independently of its containers. In Chapter 15, Redis and a background worker will add asynchronous operations to this same authenticated, database-backed API.

Chapter 15 — Add Redis and an Asynchronous Worker

A deploy request can return in milliseconds while a separate service does the work. In this chapter, you keep DeployBoard's PostgreSQL records and authenticated API from Chapter 14, then add an auditable background-job path that survives ordinary container recreation and explains exactly where recovery guarantees stop.

CHAPTER PROMISE You will submit a protected background check, observe its durable status, interrupt its worker, and verify that a replay does not repeat a completed database update. The system is an educational single-host implementation, not an HA or exactly-once queue.

Starting point: Chapter 14's complete DeployBoard source, PostgreSQL 17 database, Alembic revision 0001, and two distinct long-lived API tokens. All additions and replacements are printed in this chapter; the companion ZIP contains the complete canonical project tree, including unchanged Chapter 14 files.

15.1 Choose a Genuine Background-Work Requirement

DeployBoard already records deployments, their status transitions, and their environments. A release audit is a natural next task: an operator asks DeployBoard to assemble a deployment summary while the HTTP request returns immediately. The summary is deliberately local and deterministic, so you can test delivery semantics without introducing external APIs or payment services.

New API contract: POST /v1/deployments/{id}/checks?key=<unique-key> requires an operator token and returns HTTP 202 with a job ID. GET /v1/checks/{uuid} accepts viewer or operator credentials and reports queued, running, succeeded, or failed. Posting again with the same deployment ID and key returns the same job, not a second job.

The key is chosen by the caller. Reusing it for the same deployment means "this is the same operation." A different key means a new intentional check.

The job is stored in PostgreSQL before any Redis message is sent.

Action	System of record	Observed proof
Submit a check	PostgreSQL background_jobs	202 and stable job UUID
Deliver work	Redis Stream + consumer group	Stream entry and pending ownership
Finish a check	PostgreSQL background_jobs	Succeeded row with summary
Replay after a crash	PostgreSQL final-state check	No second completion or attempt

15.2 Introduce Redis and Internal Service Networks

Redis provides fast message transport. PostgreSQL remains authoritative for deployments and job outcomes. The publisher scans committed jobs, writes their identifiers to a Redis Stream, and the worker consumes and acknowledges each entry only after the database transaction has completed. This separation is intentional: if Redis is temporarily down, the API can still record the request in PostgreSQL.

Figure 15.1 shows the two distinct responsibilities: Redis wakes up workers; PostgreSQL preserves business facts. No database or Redis port is published on the host. Only the API binds to 127.0.0.1:8000 for this local workshop.

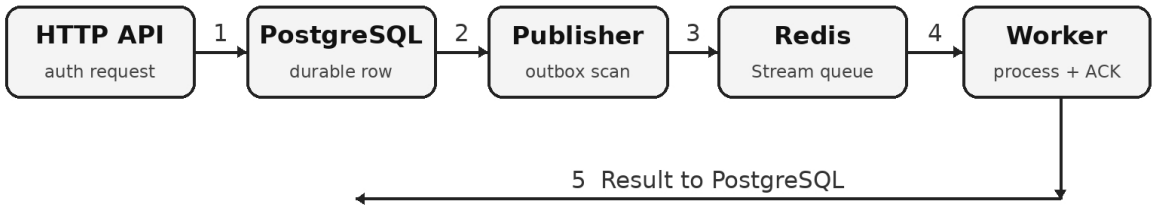


Figure 15.1 — Durable work requests and Redis Stream delivery. Redis is a transport, not the system of record.

Create the environment and Compose configuration

Copy Chapter 14's project directory, preserving app/, migrations/, tools/, and the existing .env. Replace the Compose file with the full configuration below. Its project name intentionally stays deployboard-ch14, preserving the existing PostgreSQL named volume when you upgrade in place. If you are starting with the included Chapter 15 source kit, copy .env.example to

.env and fill the three values with distinct long random secrets before running Compose. Never commit .env.

FILE compose.yaml

```
name: deployboard-ch14
services:
 db:
  image: postgres:17
  environment:
   POSTGRES_USER: deployboard
   POSTGRES_DB: deployboard
   POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in
.env}
  volumes: [dbdata:/var/lib/postgresql/data]
  healthcheck:
   test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
   interval: 5s
   timeout: 3s
   retries: 10
   start_period: 15s
  restart: unless-stopped
  networks: [private]

 redis:
  image: redis:7.4.11-bookworm
  command: ["redis-server", "--appendonly", "yes", "--appendfsync", "everysec"]
  volumes: [redisdata:/data]
  healthcheck:
   test: ["CMD", "redis-cli", "ping"]
   interval: 5s
   timeout: 3s
   retries: 10
  restart: unless-stopped
  networks: [private]

 api:
  build: .
  image: deployboard:ch15
  ports: ["127.0.0.1:8000:8000"]
  environment: &app_env
   DB_HOST: db
   DB_NAME: deployboard
   DB_USER: deployboard
   DB_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in .env}
   DEPLOYBOARD_VIEWER_TOKEN: ${DEPLOYBOARD_VIEWER_TOKEN:?set viewer
token}
   DEPLOYBOARD_OPERATOR_TOKEN: ${DEPLOYBOARD_OPERATOR_TOKEN:?set
operator token}
```

```
    REDIS_HOST: redis
  depends_on:
    db: {condition: service_healthy}
  healthcheck:
    test: ["CMD", "python", "-c", "import urllib.request;
urllib.request.urlopen('http://127.0.0.1:8000/ready')"]
    interval: 10s
    timeout: 3s
    retries: 3
    start_period: 15s
  restart: unless-stopped
  networks: [private]

 publisher:
  image: deployboard:ch15
  command: ["python", "-m", "app.publisher"]
  environment: *app_env
  depends_on:
    db: {condition: service_healthy}
    redis: {condition: service_healthy}
  restart: unless-stopped
  networks: [private]

 worker:
  image: deployboard:ch15
  command: ["python", "-m", "app.worker"]
  environment:
    <<: *app_env
    DEPLOYBOARD_FAIL_FIRST: ${DEPLOYBOARD_FAIL_FIRST:-0}
  depends_on:
    db: {condition: service_healthy}
    redis: {condition: service_healthy}
  stop_grace_period: 15s
  restart: unless-stopped
  networks: [private]

 migrate:
  image: deployboard:ch15
  profiles: [tools]
  environment: *app_env
  depends_on:
    db: {condition: service_healthy}
  command: ["alembic", "upgrade", "head"]
  networks: [private]

volumes:
 dbdata:
 redisdata:
```

```
networks:
  private: {driver: bridge}
```

WHY THE REDIS VOLUME MATTERS The Redis command enables append-only persistence and a named data volume. With appendfsync everysec, a power loss can still lose recently acknowledged Redis writes. Redis persistence is not a backup or a transaction shared with PostgreSQL.

15.3 Define the Task Payload and Worker Contract

A stream entry needs only the durable job UUID. Avoid copying deployment details, bearer tokens, or secrets into the queue: the worker loads current data from PostgreSQL and writes one bounded text result. The durable ledger records job status, attempt count, error, publication time, and the request key.

Extend app/db.py

Keep its existing deployments and status_events definitions unchanged, then append the following SQLAlchemy table declaration. This block is already included in the full app/db.py inside the companion source kit.

APPEND TO app/db.py

```
# Chapter 15 durable job ledger. Stream delivery is only a wake-up signal.
background_jobs = Table(
    "background_jobs", metadata,
    Column("id", String(36), primary_key=True),
    Column("deployment_id", Integer,
           ForeignKey("deployments.id", ondelete="CASCADE"), nullable=False),
    Column("request_key", String(64), nullable=False),
    Column("status", String(16), nullable=False),
    Column("attempts", Integer, nullable=False, default=0),
    Column("result", String(200), nullable=True),
    Column("last_error", String(200), nullable=True),
    Column("published_at", DateTime(timezone=True), nullable=True),
    Column("started_at", DateTime(timezone=True), nullable=True),
    Column("created_at", DateTime(timezone=True),
           server_default=func.now(), nullable=False),
    UniqueConstraint("deployment_id", "request_key", name="uq_job_request"),
)
```

Add one Alembic migration

Revision 0002 follows Chapter 14’s 0001 schema. Run Alembic, not metadata.create_all(), against PostgreSQL. Existing deployment rows

remain intact.

FILE migrations/versions/0002_jobs.py

```
"""Add a durable, idempotent background-check ledger."""
from alembic import op
import sqlalchemy as sa

revision = "0002"
down_revision = "0001"
branch_labels = None
depends_on = None


def upgrade() -> None:
    op.create_table(
        "background_jobs",
        sa.Column("id", sa.String(36), primary_key=True),
        sa.Column("deployment_id", sa.Integer,
                  sa.ForeignKey("deployments.id", ondelete="CASCADE"),
                  nullable=False),
        sa.Column("request_key", sa.String(64), nullable=False),
        sa.Column("status", sa.String(16), nullable=False),
        sa.Column("attempts", sa.Integer, nullable=False, server_default="0"),
        sa.Column("result", sa.String(200)),
        sa.Column("last_error", sa.String(200)),
        sa.Column("published_at", sa.DateTime(timezone=True)),
        sa.Column("started_at", sa.DateTime(timezone=True)),
        sa.Column("created_at", sa.DateTime(timezone=True),
                  nullable=False, server_default=sa.func.now()),
        sa.UniqueConstraint("deployment_id", "request_key", name="uq_job_request"),
    )
    op.create_index("ix_jobs_state", "background_jobs", ["status"])


def downgrade() -> None:
    op.drop_index("ix_jobs_state", table_name="background_jobs")
    op.drop_table("background_jobs")
```

DATA BOUNDARY Deleting a deployment cascades to its associated job rows. This demo assumes checks are disposable metadata; an audit-regulated deployment would instead retain independent immutable audit records.

15.4 Write a Minimal Worker and Queue Integration

The HTTP route writes a database job. A publisher service then writes the UUID to Redis. This is a small database-backed outbox pattern: the publisher can retry after Redis downtime, and duplicate publication is harmless because workers consult the job ledger.

Install and pin the Redis client

FILE requirements.txt

```
fastapi==0.128.2
uvicorn==0.48.0
pydantic==2.13.5
SQLAlchemy==2.0.44
alembic==1.17.1
psycopg[binary]==3.2.12
redis==6.4.0
```

Define the stream operations

FILE app/stream.py

```
"""Redis Streams transport; messages contain only durable job IDs."""
import redis

STREAM = "deployboard:checks"
GROUP = "check-workers"


def connect(host="redis", port=6379):
    return redis.Redis(host=host, port=port, decode_responses=True,
                       socket_connect_timeout=3, socket_timeout=4,
                       health_check_interval=30)


def ensure_group(client):
    try:
        client.xgroup_create(STREAM, GROUP, id="0", mkstream=True)
    except redis.exceptions.ResponseError as exc:
        if "BUSYGROUP" not in str(exc):
            raise


def publish(client, job_id):
    # Do NOT trim entries that might still be pending or required for replay.
    return client.xadd(STREAM, {"job_id": job_id})
```

Publish only committed jobs

FILE app/publisher.py

```
"""Publish committed database jobs to Redis; safe to run repeatedly."""
import argparse
import logging
```

```
import os
import time

import redis

from .jobs import Jobs
from .main import make_engine
from .settings import Settings
from .stream import connect, ensure_group, publish, STREAM

logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")
LOG = logging.getLogger("publisher")


def dispatch_once(jobs, client, rescue=False):
    ensure_group(client)
    candidates = jobs.rescue_candidates() if rescue else jobs.unpublished()
    published = 0
    for job_id in candidates:
        if client.xlen(STREAM) >= 2000:
            LOG.warning("Stream cap reached; dispatch paused")
            break
        publish(client, job_id)
        jobs.mark_published(job_id)
        published += 1
    return published


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--rescue", action="store_true",
                        help="Republish unfinished jobs once, after investigating")
    args = parser.parse_args()
    jobs = Jobs(make_engine(Settings.from_env()))
    client = connect(host=os.getenv("REDIS_HOST", "redis"))
    if args.rescue:
        LOG.info("Republished %s unfinished jobs", dispatch_once(jobs, client, True))
        return
    while True:
        try:
            count = dispatch_once(jobs, client)
            if count:
                LOG.info("Published %s committed jobs", count)
        except (redis.RedisError, OSError) as exc:
            LOG.warning("Redis unavailable; committed jobs remain in PostgreSQL: %s", exc)
        time.sleep(2)
```

```
if __name__ == "__main__":
    main()
```

In app/main.py preserve the authenticated Chapter 14 CRUD routes. Add the two new endpoints shown next immediately before the final return api. Also import UUID from uuid and Jobs from app.jobs, and change the application version to 0.15.0. The ZIP provides the resulting complete app/main.py.

ADD TO app/main.py — two protected routes

```
@api.post("/v1/deployments/{item_id}/checks", status_code=202)
def request_check(
    item_id: int,
    _: Annotated[str, Depends(require_operator)],
    request: Request,
    key: Annotated[str, Query(min_length=8, max_length=64)],
) -> dict:
    """Idempotent request: the publisher delivers it asynchronously."""
    repo = request.app.state.store
    if repo is None:
        raise HTTPException(503, detail="Database is not configured")
    job = Jobs(repo.engine).submit(item_id, key)
    if job is None:
        raise HTTPException(404, detail="Deployment not found")
    return job

@api.get("/v1/checks/{job_id}")
def check_status(
    job_id: UUID,
    _: Annotated[str, Depends(require_viewer)],
    request: Request,
) -> dict:
    repo = request.app.state.store
    if repo is None:
        raise HTTPException(503, detail="Database is not configured")
    job = Jobs(repo.engine).get(str(job_id))
    if job is None:
        raise HTTPException(404, detail="Check not found")
    return job
```

WHY THE API DOES NOT CALL REDIS The request records a queued row and returns 202 even during a broker outage. The publisher is responsible for eventual delivery. The system is not an exactly-once queue; publication can happen twice if a publisher crashes between XADD and marking a job published.

15.5 Understand Task Delivery, Retries, and Failed-Job Recovery

Redis consumer groups track messages that were delivered but not acknowledged. A worker reads with XREADGROUP, performs the work, commits to PostgreSQL, and then calls XACK. If the worker disappears before acknowledgment, XAUTOCLAIM can move an idle pending entry to another consumer. A fresh delivery is not proof that the task is new.

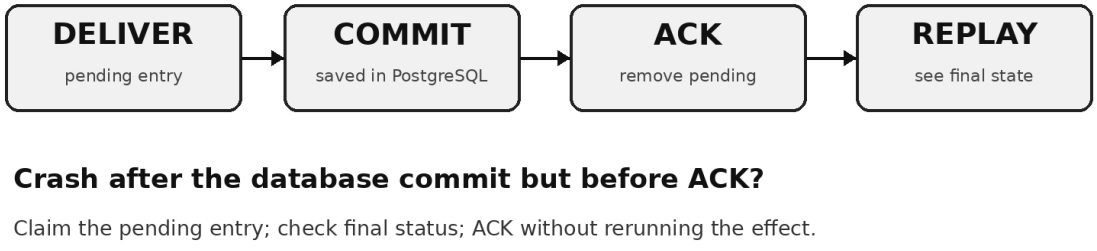


Figure 15.2 — Commit first, acknowledge second. A replay must consult persisted job state.

The complete worker process

FILE app/worker.py

```
"""At-least-once delivery within retained Redis stream + durable job ledger."""
import logging
import os
import socket
import time

import redis

from .jobs import Jobs
from .main import make_engine
from .settings import Settings
from .stream import GROUP, STREAM, connect, ensure_group

logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")
LOG = logging.getLogger("worker")
IDLE_MS = 45000  # longer than this lab's expected task runtime


def process_one(jobs, client, message_id, fields, simulate_first_failure=False):
    job_id = fields.get("job_id")
    if not job_id:
        LOG.error("Malformed stream entry %s", message_id)
```

```
        client.xack(STREAM, GROUP, message_id)
        return "malformed"
    state = jobs.claim(job_id)
    if state == "DONE":
        client.xack(STREAM, GROUP, message_id)
        return "already-final"
    if state == "BUSY":
        # Another active entry owns the same job; original remains pending.
        client.xack(STREAM, GROUP, message_id)
        return "duplicate-in-flight"
    try:
        row = jobs.get(job_id)
        if simulate_first_failure and row["attempts"] == 1:
            raise RuntimeError("Deliberate first-attempt failure")
        time.sleep(0.3)  # replace with a bounded real check in later work
        jobs.succeed(job_id)
        # Crash after commit but before XACK => replay returns already-final.
        client.xack(STREAM, GROUP, message_id)
        LOG.info("job=%s status=succeeded", job_id)
        return "succeeded"
    except Exception as exc:
        exhausted = jobs.fail_attempt(job_id, str(exc))
        if exhausted:
            client.xack(STREAM, GROUP, message_id)
        # Otherwise leave pending: idle reclaim will retry, not a busy spin.
        LOG.warning("job=%s failed=%s final=%s", job_id, exc, exhausted)
        return "failed" if exhausted else "retry-pending"


def work_cycle(jobs, client, consumer, simulate_first_failure=False):
    ensure_group(client)
    claimed = client.xautoclaim(STREAM, GROUP, consumer,
                                min_idle_time=IDLE_MS,
                                start_id="0-0", count=10)
    messages = claimed[1] if len(claimed) > 1 else []
    if not messages:
        deliveries = client.xreadgroup(GROUP, consumer,
                                       {STREAM: ">"}, count=1, block=1000)
        messages = deliveries[0][1] if deliveries else []
    for message_id, fields in messages:
        process_one(jobs, client, message_id, fields, simulate_first_failure)
    return len(messages)


def main():
    jobs = Jobs(make_engine(Settings.from_env()))
    client = connect(host=os.getenv("REDIS_HOST", "redis"))
    consumer = socket.gethostname()
```

```
    simulate = os.getenv("DEPLOYBOARD_FAIL_FIRST", "0") == "1"
    LOG.info("consumer=%s group=%s", consumer, GROUP)
    while True:
        try:
            work_cycle(jobs, client, consumer, simulate)
        except (redis.RedisError, OSError) as exc:
            LOG.warning("Redis unavailable; reconnecting: %s", exc)
            time.sleep(3)


if __name__ == "__main__":
    main()
```

The deliberate first-attempt failure can be enabled with DEPLOYBOARD_FAIL_FIRST=1. The first processing attempt stays pending; after the configured idle interval, the worker claims it again. Each durable attempt is counted. On the third failure, the job is recorded as failed and the message is acknowledged so it stops cycling.

AT-LEAST-ONCE, PRECISELY DEFINED Pending-message reclaim is at-least-once only while Redis retains the stream entry and its group state. A broker losing stream data, manual trimming, or a destructive volume deletion can defeat recovery. The PostgreSQL job ledger supports explicit rescue, but cannot itself promise instant automatic failover.

15.6 Make Task Handling Idempotent for At-Least-Once Delivery

The most important guarantee belongs to the database, not the broker. A job with final status succeeded must not be performed twice, even if Redis delivers its ID again. A transaction locks the job row, chooses whether to run, and increments its attempt count atomically. The result is committed before acknowledgment. A second message sees DONE and is safely acknowledged.

Durable job ledger and idempotent state transitions

FILE app/jobs.py

```
"""PostgreSQL is the job source of truth; retries are safe to replay."""
from datetime import datetime, timedelta, timezone
from uuid import uuid4

from sqlalchemy import insert, select, update
from sqlalchemy.engine import Engine
```

```
from .db import background_jobs, deployments

UTC = timezone.utc
MAX_ATTEMPTS = 3
LEASE_SECONDS = 45


def snapshot(row):
   return dict(id=row["id"], deployment_id=row["deployment_id"],
               status=row["status"], attempts=row["attempts"],
               result=row["result"], last_error=row["last_error"])


class Jobs:
   def __init__(self, engine: Engine):
      self.engine = engine

   def submit(self, deployment_id: int, request_key: str):
      # Row lock on parent serializes concurrent requests for the same key.
      with self.engine.begin() as db:
         parent = db.execute(select(deployments.c.id).where(
            deployments.c.id == deployment_id).with_for_update()).first()
         if parent is None:
            return None
         existing = db.execute(select(background_jobs).where(
            background_jobs.c.deployment_id == deployment_id,
            background_jobs.c.request_key == request_key
         )).mappings().first()
         if existing is not None:
            return snapshot(existing)
         job_id = str(uuid4())
         db.execute(insert(background_jobs).values(
            id=job_id, deployment_id=deployment_id,
            request_key=request_key, status="queued", attempts=0))
      return self.get(job_id)

   def get(self, job_id: str):
      with self.engine.connect() as db:
         row = db.execute(select(background_jobs).where(
            background_jobs.c.id == job_id)).mappings().first()
      return snapshot(row) if row else None

   def unpublished(self, limit=20):
      with self.engine.connect() as db:
         return list(db.execute(select(background_jobs.c.id).where(
            background_jobs.c.status == "queued",
            background_jobs.c.published_at.is_(None))
            .order_by(background_jobs.c.created_at).limit(limit)).scalars())
```

```
def mark_published(self, job_id):
    with self.engine.begin() as db:
        db.execute(update(background_jobs).where(
            background_jobs.c.id == job_id,
            background_jobs.c.published_at.is_(None))
            .values(published_at=datetime.now(UTC)))

def claim(self, job_id):
    """Returns RUN, BUSY, or DONE; locks the row for one atomic decision."""
    now = datetime.now(UTC)
    with self.engine.begin() as db:
        row = db.execute(select(background_jobs).where(
            background_jobs.c.id == job_id).with_for_update()
        ).mappings().first()
        if row is None or row["status"] in ("succeeded", "failed"):
            return "DONE"
        # A duplicate stream entry cannot start a second live attempt.
        started = row["started_at"]
        if row["status"] == "running" and started is not None:
            if started.tzinfo is None:
                started = started.replace(tzinfo=UTC)
            if now - started < timedelta(seconds=LEASE_SECONDS):
                return "BUSY"
        if row["attempts"] >= MAX_ATTEMPTS:
            db.execute(update(background_jobs).where(
                background_jobs.c.id == job_id).values(
                status="failed", last_error="Retry budget exhausted"))
            return "DONE"
        db.execute(update(background_jobs).where(
            background_jobs.c.id == job_id).values(
            status="running", attempts=row["attempts"] + 1,
            started_at=now))
        return "RUN"

def succeed(self, job_id):
    with self.engine.begin() as db:
        job = db.execute(select(background_jobs).where(
            background_jobs.c.id == job_id).with_for_update()
        ).mappings().first()
        if job is None or job["status"] == "succeeded":
            return False
        if job["status"] != "running":
            return False
        dep = db.execute(select(deployments).where(
            deployments.c.id == job["deployment_id"])).mappings().first()
        result = (f"Checked {dep['name']} ({dep['environment']}) "
                  f"at version {dep['version']}") if dep else "Deployment deleted"
```

```
        db.execute(update(background_jobs).where(
            background_jobs.c.id == job_id).values(
            status="succeeded", result=result, last_error=None))
        return True

    def fail_attempt(self, job_id, reason):
        with self.engine.begin() as db:
            row = db.execute(select(background_jobs).where(
                background_jobs.c.id == job_id).with_for_update()
            ).mappings().first()
            if not row or row["status"] in ("succeeded", "failed"):
                return True
            exhausted = row["attempts"] >= MAX_ATTEMPTS
            db.execute(update(background_jobs).where(
                background_jobs.c.id == job_id).values(
                status="failed" if exhausted else "queued",
                last_error=reason[:200], started_at=None))
            return exhausted

    def rescue_candidates(self, limit=20):
        # Operator-triggered recovery following Redis data loss.
        with self.engine.connect() as db:
            return list(db.execute(select(background_jobs.c.id).where(
                background_jobs.c.status.in_(["queued", "running"]))
                .order_by(background_jobs.c.created_at).limit(limit)).scalars())
```

This lab limits the worker to a single service instance and short bounded checks. The 45-second database lease and Redis idle threshold are educational values, not general distributed locks. Before scaling to multiple workers, implement lease fencing, heartbeats, and idempotent external side effects; otherwise an old worker could complete after a replacement begins.

DO NOT CONFUSE TWO IDEMPOTENCY KEYS The API request key prevents a user from creating the same job twice. The job UUID and final-status check prevent Redis redelivery from repeating its database effect. Both protections address different duplication paths.

15.7 Set Timeouts, Queue Limits, and Shutdown Behavior

The worker spends a fraction of a second building a local summary, so a 45-second reclaim threshold is intentionally generous. Redis client connections have short connect/read timeouts; read-group blocking lasts one second; the Compose worker receives a 15-second grace period on stop. A long-running

external operation needs a larger lease, progress heartbeats, and a bounded cancellation policy.

Control	Chapter 15 setting	What it protects
Redis connection	3 s connect / 4 s read	Avoid indefinite socket hangs
Consumer read	1 s block	Frequent chance to honor shutdown
Pending reclaim	45 s idle	Recover abandoned deliveries, not instantaneous
Failed attempts	Maximum 3	Prevent endless poison-message retries
Publisher cap	2,000 retained entries	Stop unbounded demonstration growth
Compose stop grace	15 s	Allow ordinary short jobs to exit

The 2,000-entry cap counts retained history, including acknowledged entries, because this example deliberately does not trim the stream. It is a teaching guardrail, not a production retention strategy. Production queue maintenance must examine consumer-group pending state and retention needs before deleting stream entries.

15.8 Avoid Exposing Redis on the Public Interface

A private Compose bridge network carries PostgreSQL, Redis, API, publisher, and worker traffic. The Compose file does not include ports: for db or redis:. Their services are reachable by DNS name from participating containers, but are not published to the host network.

VERIFY SERVICE BOUNDARIES

```
docker compose config
docker compose ps
docker compose exec -T redis redis-cli PING
docker compose exec -T redis redis-cli XINFO GROUPS deployboard:checks
```

WHAT TO VERIFY The Redis command should print PONG. docker compose ps should show 127.0.0.1:8000->8000/tcp for the API and no host-published ports for Redis or PostgreSQL. XINFO GROUPS is available once the publisher or worker has created the consumer group.

An internal Docker network reduces accidental exposure but is not strong isolation against a compromised Docker host. Never publish Redis port

6379 for this exercise. For serious deployments, add dedicated credentials, network controls, TLS where appropriate, and managed key rotation.

15.9 Observe Jobs Through Logs and Status Updates

The API shows business state; Redis shows delivery state. Check both. A succeeded job should have exactly one durable result even if multiple messages with its ID were published. Observe publisher and worker logs without printing tokens or full environment dumps.

Build, migrate, and launch the stack

RUN FROM PROJECT DIRECTORY

```
docker compose build
docker compose up -d db redis
docker compose --profile tools run --rm migrate
docker compose up -d api publisher worker
docker compose ps
docker compose logs --tail=30 publisher worker
```

The migration command applies 0001 if needed and then 0002. Never run docker compose down -v on a project containing records you need to keep. If a previous Chapter 14 database volume is to be retained, migrate that volume deliberately and check the Compose project name before switching.

Portable operator exercise

The helper runs inside the API container so the same command works from Bash and PowerShell. It creates a deployment, submits the same request twice, polls until a final state, and prints the job UUID and result.

FILE tools/check_jobs.py

```
"""Portable Chapter 15 smoke test; run inside the API container."""
import argparse
import json
import os
import time
from urllib.error import HTTPError
from urllib.request import Request, urlopen
from uuid import uuid4

BASE = "http://127.0.0.1:8000"


def call(method, path, token, payload=None):
```

```
    data = json.dumps(payload).encode() if payload is not None else None
    req = Request(BASE + path, data=data, method=method, headers={
        "Authorization": f"Bearer {token}",
        "Content-Type": "application/json",
    })
    with urlopen(req, timeout=6) as response:
        return json.loads(response.read().decode())


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--replay", metavar="JOB_UUID",
                        help="Publish an already-completed job ID again")
    args = parser.parse_args()
    operator = os.environ["DEPLOYBOARD_OPERATOR_TOKEN"]
    viewer = os.environ["DEPLOYBOARD_VIEWER_TOKEN"]
    if args.replay:
        from app.stream import connect, publish
        previous = call("GET", f"/v1/checks/{args.replay}", viewer)
        publish(connect(os.getenv("REDIS_HOST", "redis")), args.replay)
        time.sleep(2)
        current = call("GET", f"/v1/checks/{args.replay}", viewer)
        assert current["attempts"] == previous["attempts"]
        assert current["status"] == previous["status"] == "succeeded"
        print("Replay safe:", args.replay, current["status"],
              "attempts", current["attempts"])
        return
    deployment = call("POST", "/v1/deployments", operator, {
        "name": "Async verification lab", "environment": "staging",
        "version": "15.0.0"})
    key = "chapter15-" + uuid4().hex[:12]
    path = f"/v1/deployments/{deployment['id']}/checks?key={key}"
    queued = call("POST", path, operator)
    duplicate = call("POST", path, operator)
    assert queued["id"] == duplicate["id"]
    print("Enqueued", queued["id"], "idempotency key", key)
    for _ in range(75):
        row = call("GET", f"/v1/checks/{queued['id']}", viewer)
        if row["status"] in ("succeeded", "failed"):
            print(json.dumps(row, indent=2))
            assert row["status"] == "succeeded", "See worker logs"
            return
        time.sleep(1)
    raise SystemExit("Still queued/running: check publisher and worker logs")


if __name__ == "__main__":
    main()
```

EXPECTED WORKSHOP COMMANDS

```
docker compose exec -T api python tools/check_jobs.py
docker compose logs --tail=60 publisher worker
docker compose exec -T redis redis-cli XPENDING deployboard:checks check-workers
```

EXPECTED PROOF The script prints one UUID despite two submissions and ends with status succeeded. The PostgreSQL attempts count is normally 1. XPENDING may report zero after acknowledgment; its intermediate count depends on timing. IDs and timestamps will differ across machines.

15.10 Test Redis Unavailability, Interrupted Jobs, and Replay Safety

Lab A — Broker outage

Stop Redis, then submit a job through the helper only if you adapt its polling timeout; the regular helper expects success within 75 seconds. Alternatively issue the POST directly using the documented operator route and confirm it returns 202. The job remains queued in PostgreSQL, publisher logs connection failures, and Redis receives the entry after redis is started again.

CAREFUL: NONDESTRUCTIVE OUTAGE LAB

```
docker compose stop redis
docker compose logs --tail=15 publisher worker
# Submit a check through the operator API while Redis is down.
docker compose start redis
docker compose logs --tail=30 publisher worker
```

Lab B — An interrupted worker

After sending a new job, stop the worker before it acknowledges. The Redis consumer-group pending entry can be reclaimed after at least 45 seconds of inactivity. Because the demo task is short, an interruption must be timed carefully; an easier deterministic demonstration is to enable the deliberate first-attempt failure in the .env file.

.env — EDIT THIS VALUE (NOT A TERMINAL COMMAND)

```
DEPLOYBOARD_FAIL_FIRST=1
```

RUN FROM PROJECT DIRECTORY; THEN RESET .env TO 0

```
docker compose up -d --force-recreate worker
docker compose exec -T api python tools/check_jobs.py
```

A single injected failure should show attempts=2 and then succeeded after the pending idle period. If you stop a worker while it holds a pending message, restarting it does not erase PostgreSQL or its Redis Stream. Avoid shorter reclaim times than actual task duration.

Lab C — Replay after commit

Take a UUID from an already-succeeded job and explicitly publish that ID again. The worker must acknowledge the duplicate without changing the attempts field or the persisted result.

REPLACE JOB_UUID WITH THE PRINTED SUCCEEDED ID

```
docker compose exec -T api python tools/check_jobs.py --replay JOB_UUID
```

Local automated contract tests

The next tests exercise submission permissions, request-key deduplication, completion replay, and retry exhaustion with disposable SQLite fixtures and a fake stream acknowledgment. They do not replace a Redis or PostgreSQL container integration test.

FILE tests/test_jobs.py

```
"""Local unit/integration tests; no live Redis needed for contract checks."""
from pathlib import Path
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from app.db import metadata, background_jobs
from app.jobs import Jobs
from app.main import create_app
from app.settings import Settings
from app.store import SQLStore
from app.worker import process_one
from app.stream import STREAM, GROUP

V = "test-viewer-token-of-sufficient-length"
O = "test-operator-token-of-sufficient-length"

class FakeStream:
    def __init__(self): self.acks = []
    def xack(self, stream, group, message):
        assert (stream, group) == (STREAM, GROUP)
        self.acks.append(message)


def test_submission_permissions_and_replay(tmp_path: Path):
    db = create_engine(f"sqlite+pysqlite:///{tmp_path/'jobs.db'}")
    metadata.create_all(db)  # SQLite test fixture only; Alembic owns production.
    api = create_app(Settings(V, O), SQLStore(db))
```

```
    client = TestClient(api)
    headers = {"Authorization": f"Bearer {O}"}
    ro = {"Authorization": f"Bearer {V}"}
    created = client.post("/v1/deployments", headers=headers, json={
        "name": "DeployBoard backend", "environment": "staging", "version": "1.0"})
    assert created.status_code == 201
    url = f"/v1/deployments/{created.json()['id']}/checks?key=release-check-001"
    assert client.post(url, headers=ro).status_code == 403
    first = client.post(url, headers=headers)
    second = client.post(url, headers=headers)
    assert first.status_code == second.status_code == 202
    job_id = first.json()["id"]
    assert second.json()["id"] == job_id
    jobs = Jobs(db)
    assert jobs.unpublished() == [job_id]
    assert jobs.claim(job_id) == "RUN"
    assert jobs.succeed(job_id)
    fake = FakeStream()
    # Crash-after-commit replay: same message cannot repeat DB work.
    assert process_one(jobs, fake, "1-0", {"job_id": job_id}) == "already-final"
    assert fake.acks == ["1-0"]
    assert jobs.get(job_id)["attempts"] == 1
    assert client.get(f"/v1/checks/{job_id}", headers=ro).json()["status"] == "succeeded"
    db.dispose()


def test_retry_budget_and_failed_job(tmp_path: Path):
    db = create_engine(f"sqlite+pysqlite:///{tmp_path/'retry.db'}")
    metadata.create_all(db)
    api = create_app(Settings(V, O), SQLStore(db))
    client = TestClient(api)
    headers = {"Authorization": f"Bearer {O}"}
    item = client.post("/v1/deployments", headers=headers, json={
        "name": "Retry demo", "environment": "development", "version": "2.0"}).json()
    job = Jobs(db).submit(item["id"], "manual-retry-001")
    jobs = Jobs(db)
    for attempt in range(3):
        assert jobs.claim(job["id"]) == "RUN"
        assert jobs.fail_attempt(job["id"], "simulate connection timeout") is (attempt == 2)
    assert jobs.get(job["id"])["status"] == "failed"
    assert jobs.claim(job["id"]) == "DONE"
    db.dispose()
```

RUN LOCALLY AFTER INSTALLING requirements-dev.txt

```
python -m pytest -q
```

15.11 Recreate a Failed Worker Without Losing Core Records

A failed or redeployed worker should not own the deployment records. They live in PostgreSQL. Verify that a record created before worker recreation is still returned by GET /v1/deployments/{id}, and that any in-flight job's durable state remains visible. Redis stream retention is separate from PostgreSQL volume durability.

NONDESTRUCTIVE WORKER RECREATION

```
docker compose stop worker
docker compose rm -f worker
docker compose up -d worker
docker compose ps
docker compose logs --tail=50 worker
```

In an incident involving loss of Redis Stream data, diagnose the cause, allow the old lease to expire, then republish unfinished PostgreSQL jobs deliberately. This is an operator action, not automatic proof that no event was lost. Duplicate publication is expected and must be handled by the worker.

MANUAL RESCUE AFTER INVESTIGATION

```
docker compose exec -T publisher python -m app.publisher --rescue
docker compose logs --tail=50 worker
```

NEVER ERASE THE EVIDENCE Do not run docker compose down -v, docker volume prune, or Redis FLUSHALL during a recovery exercise. Those commands can permanently destroy PostgreSQL records or pending Redis work.

15.12 Checkpoint: Verify End-to-End Background Processing

You have finished the chapter only when you can demonstrate the following acceptance evidence, not merely show a running container.

Acceptance criterion	How to prove it
Authenticated enqueue	Operator receives 202; viewer receives 403 on POST
Durable request	New background_jobs row persists across API recreation
Service isolation	Redis and PostgreSQL have no host-published ports
Successful processing	Viewer retrieves succeeded job with meaningful result
Submission deduplication	Same request key gives the same UUID
Crash-safe replay	Replaying a succeeded UUID does not increment attempts
Bounded retries	Three intentional failures produce a failed job
Broker outage behavior	Requests stay queued in PostgreSQL until dispatch resumes
Worker replacement	Deployment and completed job records remain readable

Record the commands you ran, your observed responses, Docker Compose version, Python and Redis versions, and any deviations. Redact tokens before sharing the result. If a proof fails, diagnose the first broken boundary: API authorization, PostgreSQL schema, publisher, Redis group, worker, or final database transaction.

Canonical chapter files

Files changed or added here: app/db.py (new table), app/main.py (two endpoints), app/jobs.py, app/stream.py, app/publisher.py, app/worker.py, migrations/versions/0002_jobs.py, compose.yaml, requirements.txt, tools/check_jobs.py, and tests/test_jobs.py. All earlier Chapter 14 files remain in the complete ZIP. The source archive is a convenience; all learning steps and introduced code appear in the book.

Compatibility and known boundaries

The reference lab uses Python 3.13, FastAPI 0.128.2, SQLAlchemy 2.0.44, psycopg 3.2.12, redis-py 6.4.0, PostgreSQL 17, and Redis 7.4.11. Before future reprints, retest these pinned versions and update official compatibility notes. This is a single-host demonstration with one worker, no distributed lease fencing, no multi-host failover, and no safe automatic retention policy for aged Redis Stream entries.

Fast diagnosis at the end of this workshop

Observed symptom	Inspect first	Likely next action
API returns 503 on /ready	PostgreSQL migration and DB health	Verify Alembic revision 0002 and credentials
Job remains queued	Publisher logs; Redis PING	Restore Redis service; check stream cap
Job remains running	XPENDING; worker logs; lease age	Wait for reclaim, inspect worker crash
Repeated job has new result	DB claim/finish logic	Recheck final-state idempotency transaction
A release check disappears	DB job table; Redis AOF and volumes	Preserve evidence and use controlled rescue

Chapter 16 — Deliver a Complete Local Application Stack

A release is more than five containers that happen to be running. It is a controlled collection of services with known entry points, boundaries, persistent state, and evidence that the whole system behaves as expected. In this workshop, you will turn Chapters 13–15 into the first coherent DeployBoard local release.

Your destination. *One browser address—http://127.0.0.1:8080—serves the DeployBoard landing page and routes API calls. PostgreSQL and Redis remain private; a publisher and worker handle durable background checks. You will prove the result without buying a domain or renting a server.*

Prerequisites: Docker Engine and Compose v2 (Chapters 1–4 and 11–12), the Chapter 15 DeployBoard source tree, a working Python interpreter if you need to create a new .env file, and enough free memory for PostgreSQL, Redis, Caddy, and the application. Allow approximately 2–4 GB of available RAM for comfortable local testing; actual needs vary by platform.

Version baseline: Python 3.13-slim, PostgreSQL 17, Redis 7.4.11-bookworm, and Caddy 2.11.4-alpine. These are documented examples, not a claim that later tags are identical. Chapter 8 explains image identity and release pinning.

16.1 Consolidate the API, Database, Queue, and Worker

We start from the exact Chapter 15 contract, not from a new toy service. The FastAPI application retains /health, /ready, /v1/deployments, and /v1/checks; SQLAlchemy and Alembic own relational state; the publisher moves committed jobs into Redis Streams; the worker acknowledges an entry only after durable PostgreSQL updates.

Component	Responsibility	What persists
api	Authenticated HTTP, validation, CRUD, job requests	PostgreSQL records
db	Deployments, events, job ledger	Named volume dbdata

Component	Responsibility	What persists
redis	Redis Streams delivery and pending entries	Named volume redisdata
publisher	Committed jobs → stream entries	No local writable state
worker	Claim, retry, complete, acknowledge	Completion in PostgreSQL
proxy	Single local HTTP entry point	Static landing page is read-only

Retain the Compose project name deployboard-ch14 from Chapters 14 and 15. That historical name is intentional: the default named volumes are tied to it. If you change it to deployboard-ch16 without a migration plan, Docker creates new volume names and your earlier records seem to vanish. Reuse the original .env file so database credentials continue to match its initialized data directory.

Before replacing the stack. *Stop any running Chapter 15 services with docker compose down (without -v) from the old directory. Preserve the .env file. Never run the Chapter 15 and 16 directories simultaneously against the same Compose project.*

16.2 Add a Local Reverse Proxy and Consistent Routes

Caddy now provides one deliberate entry point. Its configuration listens on container port 8080, proxies the existing API paths without stripping prefixes, and serves a small landing page for all other routes. That avoids rewriting DeployBoard endpoint paths or inventing a new browser-facing API.

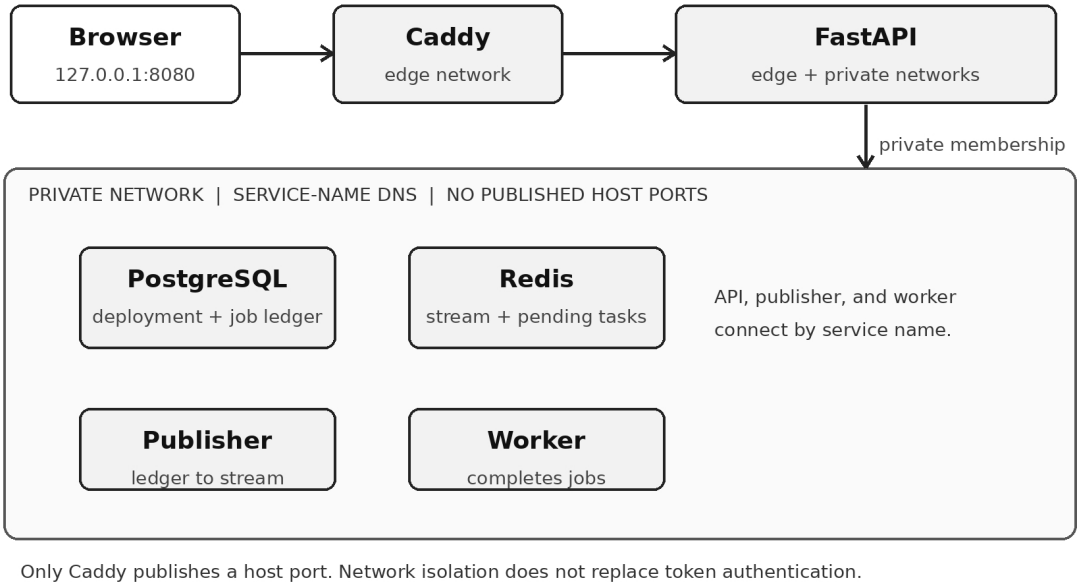


Figure 16.1 — Only the proxy publishes a localhost host port; the application spans edge and private networks.

Listing 16.1 — proxy/Caddyfile (complete)

```
:8080 {
    log {
        output stdout
        format console
    }

    @api path /health /ready /v1/* /docs /openapi.json
    handle @api {
        reverse_proxy api:8000
    }

    handle {
        root * /srv
        file_server
    }
}
```

The path matcher retains /v1/... exactly. Using handle_path here would remove a prefix and break the existing FastAPI routes. The landing page is intentionally static and has no JavaScript handling for API tokens: authentication belongs in the explicit API client or verification script.

Listing 16.2 — site/index.html (complete)

```
<!doctype html>
```

```
<html lang="en">
<head>
 <meta charset="utf-8">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <title>DeployBoard — Local Release</title>
 <style>
  body { max-width: 48rem; margin: 3rem auto; padding: 0 1.25rem;
       font: 1.08rem/1.65 system-ui, sans-serif; color: #171717; }
  header { border-bottom: 3px solid #171717; margin-bottom: 2rem; }
  code { background: #eee; padding: .15em .3em; }
  a { color: #171717; }
 </style>
</head>
<body>
 <header><h1>DeployBoard</h1><p>Chapter 16 local release</p></header>
 <p>The reverse proxy and application are connected.</p>
 <p><a href="/health">Application health</a> ·
   <a href="/ready">Application readiness</a> ·
   <a href="/docs">API reference</a></p>
 <p>Protected API routes require a bearer token. Do not enter tokens
   in links or browser address bars; use the verification script.</p>
 <p>This lab serves HTTP only on <code>127.0.0.1:8080</code>.
   It is not a public deployment.</p>
</body>
</html>
```

16.3 Organize Networks into Public and Private Boundaries

A Docker network named edge connects proxy and api; a separate network named private connects api, PostgreSQL, Redis, publisher, and worker. The proxy has no private-network membership. The private network uses internal: true to restrict external connectivity at the Docker network level; its members still communicate with each other by service name.

The only published host port belongs to Caddy: 127.0.0.1:8080:8080. Neither api:8000, db:5432, nor redis:6379 is published. “EXPOSE 8000” in the Dockerfile documents an intended container port; it does not publish that port.

Boundaries are specific. *The name edge does not itself expose a service, and internal: true is not a substitute for application authentication, host firewalling, or safe daemon privileges. This is an HTTP-only loopback lab, not a hardened Internet-facing deployment.*

The complete local Compose configuration

Replace the Chapter 15 compose.yaml with the following file. This is an explicit full listing rather than a patch: its project name, volume keys, and application environment agree with the previous chapter. It also supplies the new proxy and distinct networks.

Listing 16.3 — compose.yaml (complete)

```
# Local release; keep the project identity to reuse Chapter 14/15 volumes.
name: deployboard-ch14

x-app-env: &app_env
  DB_HOST: db
  DB_NAME: deployboard
  DB_USER: deployboard
  DB_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in .env}
  DEPLOYBOARD_VIEWER_TOKEN: ${DEPLOYBOARD_VIEWER_TOKEN:?set viewer
token}
  DEPLOYBOARD_OPERATOR_TOKEN: ${DEPLOYBOARD_OPERATOR_TOKEN:?set
operator token}
  REDIS_HOST: redis

services:
  db:
    image: postgres:17
    environment:
      POSTGRES_USER: deployboard
      POSTGRES_DB: deployboard
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:?set POSTGRES_PASSWORD in
.env}
    volumes:
      - dbdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
      interval: 5s
      timeout: 3s
      retries: 10
      start_period: 15s
    restart: unless-stopped
    networks: [private]

  redis:
    image: redis:7.4.11-bookworm
    command: ["redis-server", "--appendonly", "yes", "--appendfsync", "everysec"]
    volumes:
      - redisdata:/data
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
```

```
    timeout: 3s
    retries: 10
  restart: unless-stopped
  networks: [private]

 api:
  build: .
  image: deployboard:ch16
  environment: *app_env
  depends_on:
    db:
      condition: service_healthy
  healthcheck:
    test: ["CMD", "python", "-c", "import urllib.request;
urllib.request.urlopen('http://127.0.0.1:8000/ready', timeout=3)"]
    interval: 10s
    timeout: 4s
    retries: 3
    start_period: 20s
  restart: unless-stopped
  networks: [edge, private]
  # No host port: proxy is the only host-facing service.

 publisher:
  image: deployboard:ch16
  command: ["python", "-m", "app.publisher"]
  environment: *app_env
  depends_on:
    db: {condition: service_healthy}
    redis: {condition: service_healthy}
  restart: unless-stopped
  networks: [private]

 worker:
  image: deployboard:ch16
  command: ["python", "-m", "app.worker"]
  environment:
    <<: *app_env
    DEPLOYBOARD_FAIL_FIRST: ${DEPLOYBOARD_FAIL_FIRST:-0}
  depends_on:
    db: {condition: service_healthy}
    redis: {condition: service_healthy}
  stop_grace_period: 15s
  restart: unless-stopped
  networks: [private]

 proxy:
  image: caddy:2.11.4-alpine
```

```
    ports:
      - "127.0.0.1:8080:8080"
    volumes:
      - ./proxy/Caddyfile:/etc/caddy/Caddyfile:ro
      - ./site:/srv:ro
    depends_on:
      api: {condition: service_healthy}
    restart: unless-stopped
    networks: [edge]

  migrate:
    image: deployboard:ch16
    profiles: [tools]
    environment: *app_env
    depends_on:
      db: {condition: service_healthy}
    command: ["alembic", "upgrade", "head"]
    networks: [private]

volumes:
  dbdata:
  redisdata:

networks:
  edge:
    driver: bridge
  private:
    driver: bridge
    internal: true
```

The api service joins both networks and has no ports key. Because the proxy is on edge, it can reach api:8000 by service name but cannot resolve db or redis on the private network. The publisher and worker do not join edge and have no host-published endpoints.

16.4 Define Startup Dependencies, Readiness, and Recovery Limits

Docker Compose waits for db and redis health checks where service_healthy is declared. The API becomes healthy only when /ready succeeds, which in this application checks authentication configuration and a real database query. The proxy begins after API health succeeds. This improves startup predictability; it does not create a self-healing orchestrator.

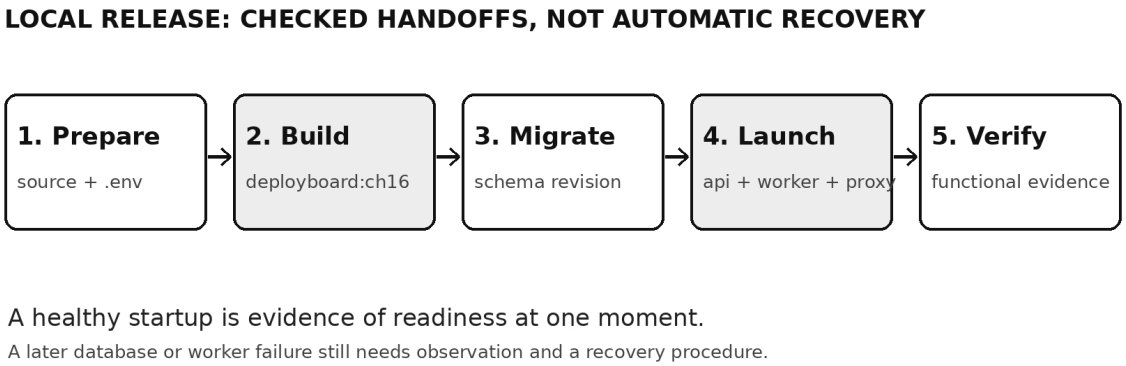


Figure 16.2 — A measured local-release sequence. Dependency checks gate startup; they do not replace incident recovery.

Start in a sequence that can be diagnosed

Command sequence 16.1 — Validate, build, migrate, and launch

```
docker compose config -q
docker compose build api
docker compose up -d db redis
docker compose run --rm migrate
docker compose up -d api publisher worker proxy
docker compose ps
```

Run the migration before launching application traffic. If an earlier chapter already applied the schema revisions, Alembic upgrade head is idempotent at the revision level and should report that no further migration is required. When you see an unhealthy service, inspect its logs before forcing restarts.

Diagnostic sequence 16.2 — Observe a failed start

```
docker compose logs --tail 40 db redis
docker compose logs --tail 50 api publisher worker proxy
docker compose ps
```

Restart policies such as unless-stopped can restart an exited container, but they do not guarantee that its dependencies recovered, that an unhealthy container will be restarted, or that an interrupted job is automatically replayed correctly. Chapter 15’s PostgreSQL job ledger and Redis reclaim logic provide application-level recovery mechanisms.

16.5 Separate Development Configuration from Runtime Images

The final application image is assembled from the unchanged Chapter 15 Dockerfile, requirements, app/, migrations/, and tools/. The proxy mounts its Caddyfile and static page read-only; those files are not baked into the Python image. No source bind mount hides application code at runtime. The .env file remains on the host and is used for Compose interpolation and service environment injection.

If you are continuing Chapter 15, keep your existing .env and skip generation. If starting from the book's printed files in a new, empty workspace, create it once using the following helper. Never regenerate it over an initialized database; a changed POSTGRES_PASSWORD environment value does not magically rotate the database user's stored password.

Listing 16.4 — tools/make_env.py (complete)

```
"""Create strong local lab credentials once; never overwrite existing .env."""
from pathlib import Path
import os
import secrets

path = Path(".env")
if path.exists():
    raise SystemExit(".env already exists; keep existing database credentials")
values = {
    "POSTGRES_PASSWORD": secrets.token_urlsafe(24),
    "DEPLOYBOARD_VIEWER_TOKEN": secrets.token_urlsafe(32),
    "DEPLOYBOARD_OPERATOR_TOKEN": secrets.token_urlsafe(32),
    "DEPLOYBOARD_FAIL_FIRST": "0",
}
with path.open("x", encoding="utf-8") as output:
    for name, value in values.items():
        output.write(f"{name}={value}\n")
if os.name == "posix":
    path.chmod(0o600)
print("Created local .env; credentials were not printed")
```

New-workspace setup (skip if .env already exists)

```
python tools/make_env.py
docker compose config -q
```

This helper creates long random local lab tokens, does not print them, and refuses to overwrite an existing .env file. On POSIX platforms it requests owner-only permissions; on Windows, review the directory ACLs. The Compose environment still exposes credentials to users with sufficient

Docker daemon privileges. Later security chapters replace these conveniences with stronger secrets handling.

16.6 Test Browser-to-API and API-to-Database Paths

Open http://127.0.0.1:8080 in a browser. The heading “DeployBoard” and links to health and readiness establish the browser-to-proxy path. Open /health and /ready; each should return HTTP 200 and JSON. The API routes stay protected: GET /v1/deployments without a bearer token must return HTTP 401.

Prove the entire HTTP contract through the proxy

The built-in Python verification script runs inside the API container but deliberately targets http://proxy:8080. It tests Caddy routing, API access controls, PostgreSQL-backed CRUD, Redis job dispatch, and completion. Its token values come from injected environment variables, not from a browser URL.

Listing 16.5 — tools/local_acceptance.py (complete)

```
"""Verify the local release through Caddy, using injected app tokens."""
import json
import os
import time
from urllib.error import HTTPError, URLError
from urllib.request import Request, urlopen
from uuid import uuid4

ROOT = os.getenv("API_URL", "http://proxy:8080").rstrip("/")
VIEWER = os.environ["DEPLOYBOARD_VIEWER_TOKEN"]
OPERATOR = os.environ["DEPLOYBOARD_OPERATOR_TOKEN"]


def call(method, path, token=None, payload=None):
    body = json.dumps(payload).encode() if payload is not None else None
    headers = {"Content-Type": "application/json"}
    if token:
        headers["Authorization"] = f"Bearer {token}"
    request = Request(ROOT + path, method=method, data=body, headers=headers)
    try:
        with urlopen(request, timeout=6) as response:
            raw = response.read()
            try:
```

```
            return response.status, json.loads(raw) if raw else None
        except json.JSONDecodeError:
            return response.status, raw.decode(errors="replace")
    except HTTPError as err:
        return err.code, None


def expect(status, method, path, token=None, payload=None):
    actual, data = call(method, path, token, payload)
    assert actual == status, f"{method} {path}: expected {status}, got {actual}"
    return data


def main():
    index = expect(200, "GET", "/")
    assert "DeployBoard" in index
    expect(200, "GET", "/health")
    expect(200, "GET", "/ready")
    expect(401, "GET", "/v1/deployments")
    expect(200, "GET", "/v1/deployments", VIEWER)
    payload = {"name": "Local release " + uuid4().hex[:8],
               "environment": "staging", "version": "16.0.0"}
    expect(403, "POST", "/v1/deployments", VIEWER, payload)
    record = expect(201, "POST", "/v1/deployments", OPERATOR, payload)
    deployment_id = record["id"]
    stored = expect(200, "GET", f"/v1/deployments/{deployment_id}", VIEWER)
    assert stored["name"] == payload["name"]
    key = "check16-" + uuid4().hex[:12]
    endpoint = f"/v1/deployments/{deployment_id}/checks?key={key}"
    first = expect(202, "POST", endpoint, OPERATOR)
    second = expect(202, "POST", endpoint, OPERATOR)
    assert first["id"] == second["id"], "Idempotency contract broken"
    for _ in range(45):
        job = expect(200, "GET", f"/v1/checks/{first['id']}", VIEWER)
        if job["status"] in ("succeeded", "failed"):
            assert job["status"] == "succeeded", f"Job failed: {job}"
            print("PASS: proxy, readiness, auth, CRUD, durable job", first["id"])
            print("Keep record ID", deployment_id, "for recreation check")
            return
        time.sleep(2)
    raise SystemExit("Job did not finish in 90 seconds; inspect logs")


if __name__ == "__main__":
    main()
```

Command 16.3 — Full release acceptance from inside Compose

```
docker compose exec -T api python tools/local_acceptance.py
```

Expected result, with variable identifiers: PASS: proxy, readiness, auth, CRUD, durable job <UUID>, followed by Keep record ID <N> for recreation check. A 401 without a token and a 403 when a viewer attempts to create a record are correct access-control results, not failures.

What this does not prove. *The script checks the proxy route from a container and the landing page is tested manually through the browser. Neither establishes HTTPS, public-network safety, high availability, or recovery from total host loss. Those are separate deployment and operations concerns.*

16.7 Test the Background-Job Lifecycle

The release verification creates a staging deployment, sends the same idempotency key twice, confirms the same job identifier is returned, then polls its status until succeeded or failed. That is a meaningful test of PostgreSQL → publisher → Redis → worker → PostgreSQL, rather than merely checking that the worker process exists.

Command 16.4 — Observe the job ledger and stream

```
docker compose logs --tail 80 publisher worker
docker compose exec -T redis redis-cli XLEN deployboard:checks
docker compose exec -T db psql -U deployboard -d deployboard -c "SELECT id, status, attempts
FROM background_jobs ORDER BY created_at DESC LIMIT 5;"
```

The inherited app/stream.py defines STREAM as deployboard:checks, so this command reads the correct stream. XLEN measures the length of the stream; the PostgreSQL status query and acceptance script prove whether a job actually completed.

Test the boundary without destroying data

Command 16.5 — Manual worker interruption and restart

```
docker compose stop worker
docker compose ps
docker compose start worker
docker compose logs --tail 35 worker
```

Stopping the worker is safe for this controlled local test, but do not infer that every in-flight task will finish immediately on restart. Chapter 15’s lease and replay handling must recover unfinished entries after the idle threshold. Do not run arbitrary prune commands to “repair” a stuck worker.

16.8 Document the Full Application File Tree

The following is the complete Chapter 16 deliverable. “Retained” means byte-for-byte inherited from Chapter 15; its full source appeared in that chapter and is included again in the accompanying complete source ZIP. “New” or “Updated” means Chapter 16 supplies a complete listing here. All files are inside this book’s chapter sequence; no external source repository is required.

```
deployboard/.dockerignore
deployboard/.env.example
deployboard/.gitignore
deployboard/Dockerfile
deployboard/START_HERE.txt
deployboard/alembic.ini
deployboard/app/__init__.py
deployboard/app/db.py
deployboard/app/jobs.py
deployboard/app/main.py
deployboard/app/models.py
deployboard/app/publisher.py
deployboard/app/security.py
deployboard/app/settings.py
deployboard/app/store.py
deployboard/app/stream.py
deployboard/app/worker.py
deployboard/compose.yaml
deployboard/migrations/env.py
deployboard/migrations/versions/0001_initial.py
deployboard/migrations/versions/0002_jobs.py
deployboard/proxy/Caddyfile
deployboard/requirements-dev.txt
deployboard/requirements.txt
deployboard/site/index.html
deployboard/tests/test_api.py
deployboard/tests/test_jobs.py
deployboard/tools/check_jobs.py
deployboard/tools/check_ready.py
deployboard/tools/local_acceptance.py
deployboard/tools/make_env.py
deployboard/tools/seed.py
deployboard/tools/smoke.py
deployboard/tools/verify_record.py
```

New or updated: compose.yaml, proxy/Caddyfile, site/index.html, tools/make_env.py, tools/local_acceptance.py, tools/verify_record.py, .dockerignore, and START_HERE.txt. All other application, migrations, tests, and dependency files remain those printed in the preceding chapters.

Listing 16.6 — tools/verify_record.py (complete)

```
"""After recreation, prove an existing deployment ID is still in PostgreSQL."""
import os
import sys
from urllib.request import Request, urlopen

if len(sys.argv) != 2 or not sys.argv[1].isdigit():
    raise SystemExit("Usage: python tools/verify_record.py DEPLOYMENT_ID")
root = os.getenv("API_URL", "http://proxy:8080").rstrip("/")
request = Request(root + "/v1/deployments/" + sys.argv[1], headers={
    "Authorization": "Bearer " + os.environ["DEPLOYBOARD_VIEWER_TOKEN"]
})
with urlopen(request, timeout=8) as response:
    payload = response.read().decode("utf-8")
    assert response.status == 200
    print("PASS: existing deployment survived recreation", payload)
```

Listing 16.7 — .dockerignore (complete)

```
.git
__pycache__/
*.pyc
.pytest_cache/
.env
.env.*
!.env.example
tests/
*.zip
*.db

site/
proxy/
```

The supplied ZIP contains a complete canonical source tree, including unchanged Chapter 15 files. In a printed-book workflow, build that same tree from the complete earlier listings, then use the Chapter 16 replacements above. You must still create a local .env file as directed; intentionally, the book and ZIP contain no working secrets.

16.9 Rebuild the Stack from Only the Book's Instructions

This is the reproducibility test. Copy the Chapter 15 canonical application files into a new project folder. Add the new proxy/, site/, and tools/ files; replace compose.yaml and .dockerignore; copy the original .env from the

Chapter 15 workspace. Since the two folders use the same Compose project name, stop the old stack before launching the new one.

12. Stop the old local stack using docker compose down, never down -v.
13. Assemble the exact Chapter 16 tree from the preceding listings (or the provided offline ZIP).
14. Preserve the previously used .env file and retain the Compose project name.
15. Run config validation, build, database startup, Alembic migration, and service startup as in 16.4.
16. Run the acceptance script and retain the printed deployment ID.
17. Stop and recreate the containers without deleting named volumes.
18. Verify the previously created deployment ID remains accessible.

Command 16.6 — Persistence check (replace 123 with your saved ID)

```
docker compose down
docker compose up -d db redis
docker compose run --rm migrate
docker compose up -d api publisher worker proxy
docker compose exec -T api python tools/verify_record.py 123
```

Use the actual record ID reported by local_acceptance.py; “123” is only a placeholder. A successful HTTP 200 and a matching deployment record after recreation prove persistence in this exercise. If the request returns 404, first check the Compose project name, the named volume identity, and whether you used the correct record ID.

Avoid the destructive shortcut. *docker compose down preserves named volumes by default; docker compose down -v deletes Compose-managed volumes. Do not run the latter during this chapter.*

16.10 Record the Baseline Before Security Hardening

A release baseline is an evidence packet, not just a claim that “it works on my machine.” Record a short set of observable facts before later chapters introduce least privilege, file-mounted credentials, and supply-chain verification.

Evidence	How to capture it	Acceptable result
Compose config	docker compose config -q	No validation error

Evidence	How to capture it	Acceptable result
Application identity	docker image inspect deployboard:ch16	Image ID recorded
Startup	docker compose ps	Expected services running; db, redis, api healthy
Network exposure	docker compose ps; inspect published ports	Only proxy loopback host port
Authenticated API	tools/local_acceptance.py	Expected 401 / 403 / 200 / 201
Background jobs	Acceptance output + logs	Unique request key, finished job
Persistence	tools/verify_record.py after down/up	Original record still available

Command 16.7 — Minimal release-evidence snapshot

```
docker compose config -q
docker compose ps
docker image inspect deployboard:ch16 --format '{{.Id}}'
docker volume ls --filter name=deployboard-ch14
```

Do not paste the unredacted output of docker compose config into a public issue or book errata report: it can reveal injected environment credentials. Likewise, do not publish your .env file, Docker daemon endpoint, or access tokens.

Production distinction: local HTTP is not encrypted; test tokens are simple shared bearer secrets; db and redis image tags must be reviewed and eventually pinned by digest; and this is a single-host Compose design. A production-ready claim belongs to the final capstone after later chapters prove secrets, HTTPS, backups, and controlled delivery.

16.11 Checkpoint: Complete the DeployBoard Local Release

You are finished when another person can follow the book's file listings and recreate the same operational behavior without a downloaded code repository or undocumented commands.

Acceptance item	Required evidence
One predictable entry point	Browser reaches http://127.0.0.1:8080
Private data services	No published PostgreSQL/Redis ports
Service readiness	API /ready returns 200 after migrations

Acceptance item	Required evidence
Access control	Anonymous 401; viewer write 403; operator write 201
Durable job	One ID on duplicate submission; final succeeded
Database persistence	Saved deployment survives container recreation
Repeatable build	Compose validates and image rebuilds from the documented tree
Safe shutdown	docker compose down without -v leaves volumes present

If any requirement fails, capture the smallest useful evidence: command, relevant service logs, expected behavior, and actual behavior. Fix that specific gap before treating the local release as a stable base for Part V.

What you have shipped. *A locally accessible, authenticated, multi-container DeployBoard application with a persistent database and background jobs. It is ready to be hardened—not ready to be exposed directly to the public Internet.*

PART V

SECURE IMAGES, CONTAINERS, AND SECRETS

Chapter 17 — Understand Docker's Security Model and Threat Surface

DeployBoard now works as a complete local application. The next engineering decision is not which lock to add first. It is what the system exposes, who can control it, and what would happen if one component were compromised. In this chapter, you will inspect the actual Chapter 16 release without deliberately weakening it.

The security question. If an attacker can call your API, reach a database port, read a credential, or control the Docker daemon, what can they do next? Your first deliverable is an evidence-backed threat register—not a claim that a container is automatically secure.

Starting point: use the Chapter 16 DeployBoard source tree and the same Compose project name, deployboard-ch14. Services: proxy (Caddy), api (FastAPI), db (PostgreSQL 17), redis, publisher, worker, and a profile-gated migration task. Keep your existing named volumes; this chapter uses read-only inspection.

Working convention: commands assume the terminal is in the directory containing compose.yaml and that the stack is already running. Commands containing single quotes are shell examples: PowerShell and Bash both accept the Docker formatting shown. No command in this chapter prints token or password values.

17.1 Define the Assets and Trust Boundaries of DeployBoard

Begin with what deserves protection, not with a list of Docker features. DeployBoard stores deployment histories and background-job states in PostgreSQL; Redis carries event work; viewer and operator tokens authorize API activity. The host, Docker daemon, image supply chain, and local .env are part of the system too—even though they do not appear on the application diagram.

Asset / boundary	Value or consequence	Who should reach it
API and operator token	Read or change deployments	Authorized application clients
PostgreSQL volume	Historical records and job state	API, publisher, worker, authorized admin
Redis stream / volume	Queued and pending work	Publisher and worker only
Caddy port 8080	Local entry point to site and API	This machine during the lab
Docker daemon/socket	Control images, mounts, containers	Trusted host administrator only
Images and dependencies	Code executed with container privileges	Audited build and release workflow

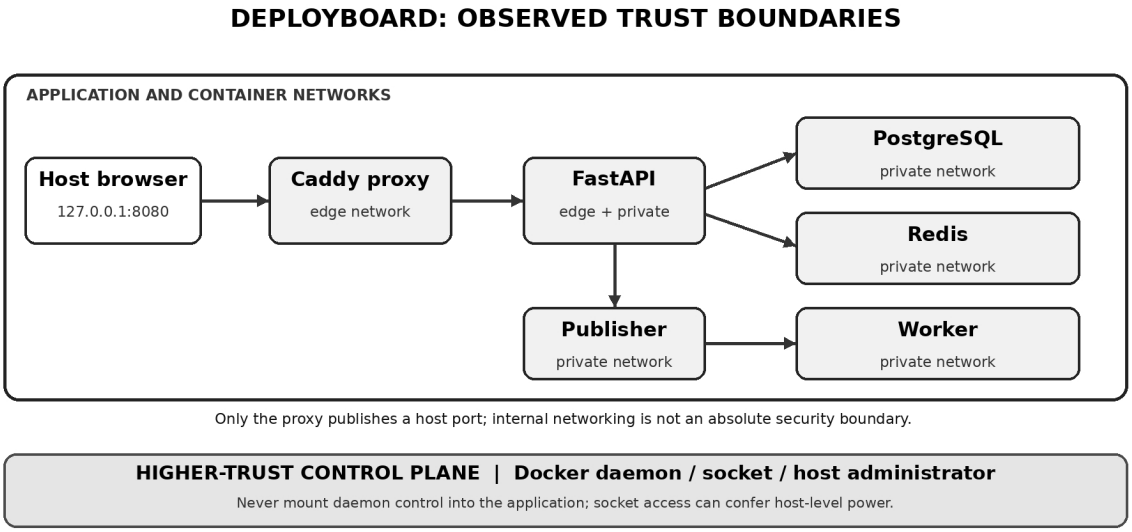


Figure 17.1 — Observe DeployBoard's application networks separately from the higher-trust Docker control plane.

Follow the paths. The proxy is on the edge network. The API joins edge and private. Database, Redis, publisher, and worker attach only to private. The Compose file publishes 127.0.0.1:8080:8080 for the proxy; this does not mean that every service is inherently isolated from the Docker host.

Inspection 17.1 — list active services without exposing the .env file

```
docker context show
```

```
docker compose ps
docker compose config --services
```

Expected evidence: the current Docker context, active service rows, and names including api, db, proxy, redis, publisher, worker, and migrate. The profile-gated migrate service may not be running. Service listing does not prove that a port is unreachable from another machine; that test requires an actual network check.

17.2 Distinguish Container Isolation from a Security Guarantee

A container uses host-kernel features to isolate processes, filesystems, networking, and other resources. Isolation reduces the ordinary visibility of other workloads, but a container is not a separate general-purpose security operating system. Host-kernel vulnerabilities, unsafe privilege grants, dangerous mounts, and exposed daemon access can undermine boundaries.

Do not confuse three claims: (1) “the service is on its own Docker network,” (2) “the service is not published to the host,” and (3) “an untrusted process cannot affect the host.” They describe different controls. The first two are evident in Compose; the third cannot be established from network configuration alone.

Three concrete boundaries to recognize

- Application boundary: a bearer token and role check can reject an unauthorized HTTP request; it does not protect Docker’s socket.
- Container boundary: a non-root runtime identity limits ordinary filesystem privileges; it does not make container code harmless.
- Host boundary: Docker daemon access can allow creation of high-impact containers and mounts; treat it like administrative power.

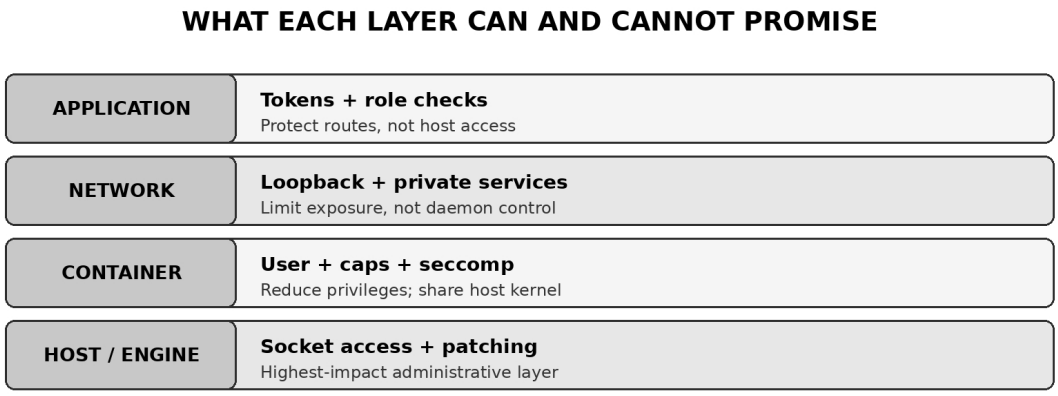


Figure 17.2 — Docker security is layered; no single layer replaces the others.

For this chapter. Read the observed baseline and record gaps. Changes such as capability dropping, read-only root filesystems, and secret-file migration belong to Chapters 18 and 19, where each change will be paired with functional tests.

17.3 Identify Risks in Base Images and Dependencies

Chapter 16 uses python:3.13-slim for the DeployBoard image, postgres:17, redis:7.4.11-bookworm, and caddy:2.11.4-alpine. The exact image digests depend on when the images were pulled or built. A familiar name is not evidence of current security, and a version tag alone is generally not an immutable identifier.

Inspect what is actually running

Inspection 17.2 — enumerate images and inspect the API image without printing environment values

```
docker compose images
docker image inspect deployboard:ch16 --format '{{.Id}}'
docker image inspect deployboard:ch16 --format '{{.Config.User}}'
docker image history --no-trunc deployboard:ch16
```

The image ID and history help identify the specific locally built artifact and its construction. History can inadvertently reveal unsafe build instructions from any image; inspect privately. Do not publish raw inspection output or assume that an image’s current label reveals all transitive dependencies.

Image review will become systematic in Chapter 20: choose trusted maintainers, use a digest when repeatability matters, review package provenance, evaluate vulnerability results, and rebuild after controlled

updates. A vulnerability finding needs context: affected component, exploitability, fixed version, exposure, and business impact.

Risk	Possible evidence	Practical response
Mutable tags	Pulled digest differs across dates	Record digests; test update builds
Unpatched dependencies	Scanner advisory references package	Assess impact and rebuild
Unknown provenance	Unclear image source/build inputs	Prefer maintained, traceable images
Too many tools in runtime	Unneeded packages or compilers	Use reviewed multi-stage builds

What is not proved. A small image is not automatically a secure image; a clean scan is not proof of safety; and a digest protects identity, not the trustworthiness of its contents.

17.4 Understand Docker Daemon and Socket Privileges

The Docker client asks a daemon to create and control containers. On a typical rootful Linux Engine installation, the daemon runs with host-level privileges. Membership in the docker group often gives a user root-level control through the daemon; being able to type docker without sudo is not evidence of least privilege. On Docker Desktop, a Linux virtual machine changes the platform boundary but does not remove the importance of trusted engine access.

Inspection 17.3 — observe current engine context and available security options

```
docker context show
docker context ls
docker info --format '{{json .SecurityOptions}}'
```

Expected evidence: the active context and a JSON-like list of reported engine security options. Exact entries vary by platform and Engine version. This is an observation, not a command to reconfigure the daemon.

The UNIX socket is commonly /var/run/docker.sock inside Linux environments. Mounting that socket into an application container allows it to issue daemon API requests and is especially dangerous for a public-facing service. A remote TCP API without appropriate authentication and transport protection creates a similar administrative exposure.

Inspection 17.4 — check for socket presence inside the API container (read-only)

```
docker compose exec -T api python -c "import os; print(os.path.exists('/var/run/docker.sock'))"
```

For the Chapter 16 baseline, expect False. A True result requires immediate review of the mount and daemon permissions; the command itself does not open or use the socket. If a command fails because the API container is not running, restore the baseline stack before continuing.

Never run this "test." Do not demonstrate Docker socket access by mounting the host filesystem, launching privileged containers, or changing daemon listening ports. Those are unnecessary and unsafe ways to verify this chapter's security baseline.

17.5 Recognize Dangerous Bind Mounts and Privileged Mode

A bind mount gives a container access to a selected host path. Read-only mounts narrow what the process can write, but they do not automatically make the mounted information harmless. A writable mount of sensitive host directories, a mounted Docker socket, or privileged mode can bypass much of the isolation people expect from a container.

DeployBoard's proxy intentionally reads ./proxy/Caddyfile and ./site as read-only mounts. PostgreSQL and Redis use named volumes for durable application state. The Chapter 16 Compose configuration does not request privileged mode or add a daemon socket mount. Record those facts as a baseline; Chapter 18 will introduce additional hardening.

Inspection 17.5 — verify the API runtime identity

```
docker compose exec -T api id
docker compose exec -T api python -c "import os; print('UID:', os.geteuid())"
```

The application Dockerfile creates the user app and sets USER app; you should see a nonzero UID. This is a valuable control, but "UID is nonzero" is not equivalent to checking filesystem permissions, Linux capabilities, image vulnerabilities, or kernel attack surface.

Interpreting runtime flags

Setting	What to look for	Security interpretation
Privileged	HostConfig.Privileged = false	Good baseline; do not enable casually
Bind / volume mounts	Destinations and RW flags	Match exactly what service needs
Read-only rootfs	Currently false in baseline	Future hardening candidate; test writes
Dropped capabilities	None explicitly configured	Review actual need in Chapter 18
Docker socket mount	Absent from service mounts	Essential control-plane separation

17.6 Inspect Exposed Ports and Internal Services

Published ports create a host access path. The Chapter 16 stack publishes only the proxy at 127.0.0.1:8080; the API, PostgreSQL, and Redis do not publish host ports. That is a sensible local learning default, but localhost binding is not a substitute for validating host firewall behavior before a public deployment.

Inspection 17.6 — list host-facing ports

```
docker compose ps
docker compose port proxy 8080
```

Expected output from the port query is 127.0.0.1:8080 (format may vary slightly). Do not expect a published host port for db on 5432 or redis on 6379. Exposing a port inside an image or accepting traffic on a Docker network is different from publishing it to the host.

Verify one allowed path and one denied path

Inspection 17.7 — from the host; PowerShell users can use curl.exe

```
curl -i http://127.0.0.1:8080/health
curl -i http://127.0.0.1:8080/v1/deployments
```

With a healthy stack, the first endpoint should return HTTP 200. The second should return HTTP 401 without a valid bearer token. If you receive an unexpected 2xx result from the protected endpoint, stop and investigate API routing and authentication. HTTP status alone does not prove that the database is unreachable from another device.

Host firewalls deserve separate testing on native Linux. Docker’s published-port rules interact with iptables or nftables and may bypass

ordinary UFW expectations; this is a real operational concern when production ports become public. The full host-level verification comes in Chapter 27.

17.7 Explain Namespaces, Capabilities, and Seccomp at a High Level

Linux provides several complementary controls. Namespaces limit what a process can see, cgroups limit or account for resources, capabilities divide traditionally broad root privileges, and seccomp filters restrict system calls. They serve different purposes; none is a complete security policy by itself.

Mechanism	What it primarily controls	Practical takeaway
PID / mount namespaces	Process and mount views	Containers see scoped process and mount trees
Network namespaces	Interfaces, addresses, routes	Each container can have an isolated network stack
User namespaces	UID and GID mappings	Root inside may map to unprivileged host UID
cgroups	CPU, memory, PIDs and accounting	Resource limits reduce some denial-of-service risk
Linux capabilities	Selected kernel privilege powers	Drop unused capabilities after compatibility testing
seccomp	Permitted system calls	Default profile reduces syscall surface

Docker normally provides a default seccomp profile on supported Linux environments unless overridden. Treat a missing explicit SecurityOpt in the Compose file as “engine default applies if enabled,” not as proof that seccomp is off or as evidence that its exact profile is known. Chapter 18 will check these defenses without guessing from configuration alone.

Engineering judgment. Do not replace meaningful application authentication with container flags. A seccomp profile cannot decide whether a particular user is allowed to edit a deployment record.

17.8 Compare Rootful, Rootless, and User Namespace Options

There are two separate questions: “Who runs the process inside the application container?” and “Under which host privileges does the Docker daemon operate?” The first is addressed by USER app in the Dockerfile. The second depends on the Docker Engine mode and host configuration.

Mode	Daemon privilege	Important tradeoff
Rootful Engine	Usually root on Linux	Common default; socket access is high-trust
Rootful + userns-remap	Daemon still root	Maps container UIDs; host path ownership can change
Rootless Engine	Daemon and containers run without host root	Reduces daemon privilege; has prerequisites/limitations
Docker Desktop	Daemon in managed Linux VM	Different host boundaries; not identical to native rootless

Rootless mode uses user namespaces to run both daemon and containers without host root privileges, subject to documented platform prerequisites. userns-remap maps container IDs while the daemon remains root. Neither should be switched on casually in the middle of a stateful Compose workshop: volume ownership, networking, and operational behavior may change.

You will evaluate whether rootless mode is appropriate in Chapter 18. For now, record your actual configuration with docker info; do not infer it from your application’s non-root UID or from the presence of Docker Desktop.

17.9 Create a Practical Docker Threat Checklist

An effective checklist names an asset, a credible misuse path, observable evidence, the current control, and a next action. “Secure Docker” is not an auditable requirement. The following register is tailored to the exact Chapter 16 Compose release; it distinguishes established baseline facts from items that still require testing.

Finding to review	Baseline evidence	Follow-up chapter
Public API abuse	Bearer tokens and viewer/operator roles; rate limit not yet defined	18 / 28
Credentials exposure	DB and API tokens passed via environment variables	19
Host / daemon takeover	No socket mount in Compose; check socket access	17 / 18
Container privilege excess	API uses app; cap-drop and read-only rootfs not set	18
Untrusted image dependencies	Versioned images; current scan not proved	20
Database exposure	db private network, no published port	17 / 27
Redis exposure	redis private network, no published port	17 / 27
Production transport	Local HTTP on loopback; no public TLS yet	28
Recoverability	Persistent volumes exist; restore drills required	29

Severity is a decision, not a decoration

Prioritize by reachability, potential impact, exploitability, and available compensating controls. A missing public TLS certificate on a deliberately localhost-only lab is not the same exposure as the same configuration on an internet-facing host. State the scope of every finding.

17.10 Checkpoint: Audit the Baseline Stack

You will now create a concise security report for the running DeployBoard services. The Python program below uses only the standard library and read-only Docker commands. It does inspect in-memory Docker JSON, which can contain secret values; critically, it prints only a safe allowlist of

fields and the names—not values—of selected sensitive environment variables. Do not replace it with unfiltered docker inspect output in screenshots or bug reports.

Listing 17.1 — tools/security_baseline.py (complete; copy exactly)

```
"""Read-only baseline assessment for the Chapter 16 DeployBoard stack.
Run from the project directory: python tools/security_baseline.py
The report shows sensitive variable NAMES, never their values.
"""
import json
import subprocess
import sys
SERVICES = ("proxy", "api", "db", "redis", "publisher", "worker")
SENSITIVE = {"DB_PASSWORD", "POSTGRES_PASSWORD",
             "DEPLOYBOARD_VIEWER_TOKEN", "DEPLOYBOARD_OPERATOR_TOKEN"}
def docker(*args):
    command = ["docker", *args]
    result = subprocess.run(command, text=True, capture_output=True)
    if result.returncode:
        message = (result.stderr.strip().splitlines() or ["command failed"])[0]
        raise RuntimeError(f"{' '.join(command[:3])}: {message}")
    return result.stdout.strip()
def read_service(service):
    ids = docker("compose", "ps", "-q", service).splitlines()
    if len(ids) != 1:
        raise RuntimeError(f"Start service {service} before auditing")
    return json.loads(docker("inspect", ids[0]))[0]
def examine(service, item):
    host = item["HostConfig"]
    config = item["Config"]
    network = item["NetworkSettings"]
    mounts = item.get("Mounts", [])
    bound = []
    for exposed, mappings in (network.get("Ports") or {}).items():
        for mapping in mappings or []:
            bound.append(f"{mapping['HostIp']}:{mapping['HostPort']}->{exposed}")
    socket = any(m.get("Destination") == "/var/run/docker.sock"
                 for m in mounts)
    keys = {line.split("=", 1)[0] for line in (config.get("Env") or [])}
    labels = sorted(keys & SENSITIVE)
    print(f"\n[{service}] image={config.get('Image', '?')}")
    print(f"  running-user={config.get('User') or '(image default)'}")
    print(f"  privileged={host.get('Privileged', False)} socket-mount={socket}")
    print(f"  read-only-rootfs={host.get('ReadonlyRootfs', False)}")
    print(f"  cap-drop={host.get('CapDrop') or 'none specified'}")
    print(f"  security-options={host.get('SecurityOpt') or 'engine defaults'}")
    print(f"  published-ports={', '.join(bound) if bound else 'none'}")
```

```
        print(f"  networks={', '.join(sorted((network.get('Networks') or {})))}")
        print(f"  sensitive-env-keys={', '.join(labels) if labels else 'none'}")
    def main():
        print("DEPLOYBOARD SECURITY BASELINE | read-only | no secret values")
        print("Docker context:", docker("context", "show"))
        for service in SERVICES:
            examine(service, read_service(service))
        print("\nManual follow-up: verify host firewall and external access.")
        print("No configuration was changed by this report.")
    if __name__ == "__main__":
        try:
            main()
        except (OSError, ValueError, RuntimeError, KeyError) as exc:
            print(f"Audit not complete: {exc}", file=sys.stderr)
            sys.exit(1)
```

Run the assessment

Inspection 17.8 — work from the Chapter 16 project directory

```
python tools/security_baseline.py
```

Expected report shape: six service blocks with their image reference, runtime user, privileged flag, socket-mount status, read-only-rootfs status, dropped-capability configuration, security-option overrides, published-port bindings, networks, and sensitive environment-variable names. Values differ across machines; the program deliberately does not print secret values.

Make the release decision with evidence

- Confirm the only host-published endpoint is proxy on 127.0.0.1:8080; anything else needs an explanation and correction.
- Confirm privileged=false and socket-mount=false for every application service. A failure is a blocker pending investigation.
- Confirm the API runs as a non-root user and the unauthenticated protected endpoint returns 401.
- Record environment-based credential injection as a known improvement for Chapter 19; do not claim secrets are protected just because .env is ignored by Git.
- Record missing capability dropping, read-only rootfs, resource restrictions, and image vulnerability evidence as hardening work, not as current guarantees.

Checkpoint artifact. Save a private one-page baseline note with the Docker context, execution date, service-to-network map, observed ports, audit results (without secrets), and a prioritized action list. The checkpoint is complete when you can explain every significant boundary and show how you verified it.

The next chapter turns this evidence into controls: non-root execution across appropriate services, narrower capabilities, carefully chosen security profiles, controlled writable paths, and realistic resource boundaries. Every change must preserve DeployBoard's working API, queue, database, and background job.

Chapter 18 — Harden Containers with Least Privilege

> *A secure container is not one with every available restriction switched on. It is one with only the privileges and write paths its workload actually needs—and evidence that the service still works.*

DeployBoard reached a complete local release in Chapter 16. Chapter 17 identified its trust boundaries. Now you will convert selected security goals into controlled changes that you can validate, measure, and undo. Keep the existing Compose project name, data volumes, and authentication contract. This chapter hardens the API, publisher, and worker first; it treats stateful services as separate engineering decisions.

FIGURE 18.1 Harden the workload, preserve the service path

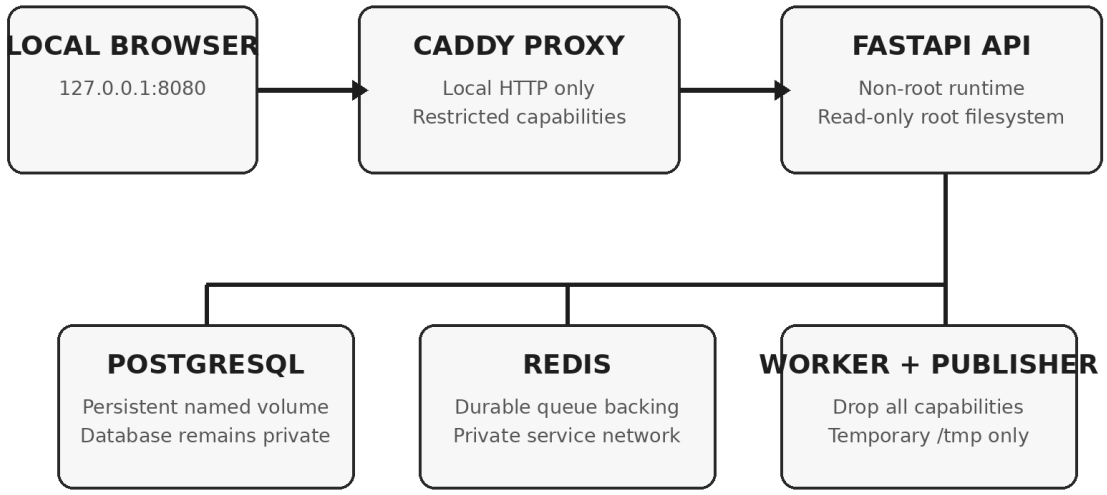


Figure 18.1 — Workload hardening is layered on top of the established DeployBoard network and data design.

THE RELEASE CONTRACT

You will add one Compose override named compose.harden.yaml, then prove that the API still answers authenticated requests, PostgreSQL records remain durable, background jobs complete, and the proxy remains

loopback-only. None of the commands below intentionally deletes volumes. You need Docker Engine/Desktop, Compose v2, Python 3, and the full Chapter 16 project files already provided in this book.

IMPORTANT BOUNDARY

This is a local, HTTP-only security lab—not a claim of internet-ready deployment. Chapters 19 and 20 handle secrets and image supply-chain checks; later chapters add public TLS, CI/CD, backups, and host hardening. Running as non-root and dropping capabilities do not make untrusted code safe by themselves.

18.1 Create and Verify Non-Root Application Users

Start with the immutable image rather than changing a live container. The Chapter 16 Dockerfile already creates an app system user, copies application files to the image with matching ownership, and selects USER app. That is a good baseline: application code can be read without making the process root. Never assume that USER app proves every service—including PostgreSQL and Caddy—runs without root; inspect each separately.

Inspect the existing Dockerfile and running user

```
docker compose build api
docker image inspect deployboard:ch16 --format "{{.Config.User}}"
docker compose exec -T api id
```

Expected: the image inspection prints app; id reports a nonzero UID. If the Compose stack is not running, start it using the Chapter 16 procedure before the exec command. The exact numeric UID can change when the underlying base image changes, so record the value actually observed rather than hard-coding it.

Key existing Dockerfile instructions

```
RUN groupadd --system app && \
    useradd --system --gid app --no-create-home app
COPY --chown=app:app app ./app
USER app
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Do not replace a working non-root image with a root image to fix a permissions error. Identify the specific required path, adjust ownership at build time, or provide a narrowly scoped writable mount. The authenticated

API remains on internal port 8000; the reverse proxy is still the only host-facing service.

18.2 Apply File Ownership and Safe Runtime Permissions

An image user, an on-disk owner, and a mount owner are three different things. COPY --chown controls ownership of files added during build. A bind mount obscures the image path beneath it and exposes the host filesystem's permissions; a named volume follows its own ownership rules. If a writable directory is genuinely necessary, create it deliberately and give the application user access only to that directory.

Inspect relevant paths without writing to them

```
docker compose exec -T api sh -c "id; ls -ld /service /service/app /tmp"
```

A permission failure is evidence to diagnose—not a reason to use chmod 777. The Chapter 16 API should not edit its source tree at runtime. PostgreSQL writes belong in dbdata and Redis append-only data belongs in redisdata. Those services retain their existing named volumes and ownership handling in this exercise.

Path or asset	Policy	Reason
/service and /service/app	Read-only at runtime	Versioned code belongs in the image
/tmp in Python services	Bounded, ephemeral tmpfs	Scratch data must not be durable
dbdata	Persistent named volume	Retain deployment and job records
redisdata	Persistent named volume	Retain Redis append-only queue state

18.3 Drop Unnecessary Linux Capabilities

Linux capabilities partition powerful kernel operations into smaller privileges. A simple Python HTTP server on port 8000, a Redis publisher, and a worker should not need capabilities such as NET_ADMIN or SYS_ADMIN. Docker already drops several dangerous capabilities by default. The additional cap_drop: [ALL] setting removes the remaining capability set from these application processes. It is a strong choice only after functional testing.

Apply the override without changing the baseline file

Create the exact file compose.harden.yaml in the root of the Chapter 16 project. Its full contents appear below. This override adds controls to selected services; Compose merges it with the original compose.yaml. Run the two files together, in that order. It is intentionally not a stand-alone Compose project.

```
# Chapter 18: an additive Compose override. Never start this file alone.
# Apply with: docker compose -f compose.yaml -f compose.harden.yaml up -d --build
x-app-hardening: &app_hardening
  read_only: true
  cap_drop:
    - ALL
  security_opt:
    - no-new-privileges:true
  tmpfs:
    - /tmp:rw,noexec,nosuid,size=32m,mode=1777
  mem_limit: 384m
  cpus: 1.0
  pids_limit: 128

services:
  api:
    <<: *app_hardening
  publisher:
    <<: *app_hardening
    mem_limit: 256m
    cpus: 0.5
  worker:
    <<: *app_hardening
    mem_limit: 256m
    cpus: 0.5
  proxy:
    # Keep the baseline proxy's existing writes/mounts until separately tested.
    cap_drop: [ALL]
    security_opt:
      - no-new-privileges:true
    mem_limit: 192m
    cpus: 0.5
    pids_limit: 96
```

The YAML anchor named app_hardening gives the Python services a common baseline and lets publisher and worker override their resource budgets. It does not alter their commands, persistent data paths, environment variables, Compose project name, or existing network memberships. The proxy also drops all capabilities because this local lab listens on the unprivileged container port 8080, but it is not yet given a blanket read-only filesystem.

Confirm the Compose merge before touching a running service

```
docker compose -f compose.yaml -f compose.harden.yaml config --quiet
python tools/hardening_audit.py
```

Expected: Compose exits with status zero and the audit prints PASS lines for the three Python services and port exposure. The audit reads the merged configuration but never prints interpolated secrets. Do not use docker

compose config without a safe output strategy if it would reveal environment values or tokens.

18.4 Use no-new-privileges and Appropriate Security Profiles

The no-new-privileges security option prevents processes from gaining additional privileges through mechanisms such as set-user-ID execution. It complements, rather than replaces, a non-root USER and capability dropping. Compose accepts no-new-privileges:true under security_opt. Keep the Docker Engine's built-in seccomp policy enabled; do not use seccomp=unconfined or privileged: true to get a lab to work.

AppArmor, SELinux, and seccomp enforcement depend on the Linux host and Docker configuration. Docker Desktop runs Linux containers inside a managed Linux VM. Avoid copying an AppArmor profile name from one Linux distribution to another: instead inspect platform support, preserve defaults, and add custom profiles only after targeted tests.

Inspect the applied runtime controls

```
python tools/runtime_inspect.py
docker compose -f compose.yaml -f compose.harden.yaml exec -T api \
  python -c "print(next(x for x in open('/proc/1/status') if x.startswith('NoNewPrivs:')).strip())"
```

The first command reports state, runtime user, ReadonlyRootfs, CapDrop, SecurityOpt, memory bytes, and PIDs limit; it does not emit environment variables. The second command runs inside the Linux container and should show NoNewPrivs: 1 when the restriction is active. If your host shell handles backslashes differently, enter that docker compose command on one line.

18.5 Introduce Read-Only Root Filesystems and Writable Mounts

A read-only root filesystem is a runtime rule. It prevents writing to the container's ordinary root filesystem, even when image layers contain writable-looking directories. Processes may still need a scratch path. For the three Python services, compose.harden.yaml sets read_only: true and mounts /tmp as a bounded tmpfs with noexec and nosuid. The byte limit is a safety boundary, not extra memory: tmpfs data still consumes the container's memory allocation.

Prove the intended difference

```
docker compose -f compose.yaml -f compose.harden.yaml exec -T api python tools/prove_writes.py
```

Expected: the script prints PASS for /tmp write/read/remove and PASS for /service/app being read-only. It distinguishes the filesystem error EROFS from a simple user-permission denial, so the test actually proves read-only enforcement. The script is shown in full later in this chapter and copied into the image by the existing Dockerfile.

Do not apply the same override blindly to db, redis, or proxy. PostgreSQL initialization and normal operation require write access; Redis append-only storage needs its data volume; and Caddy may require configuration and certificate storage paths in later HTTPS chapters. Each service deserves a separate tested write-path inventory.

FIGURE 18.2 Make write access explicit

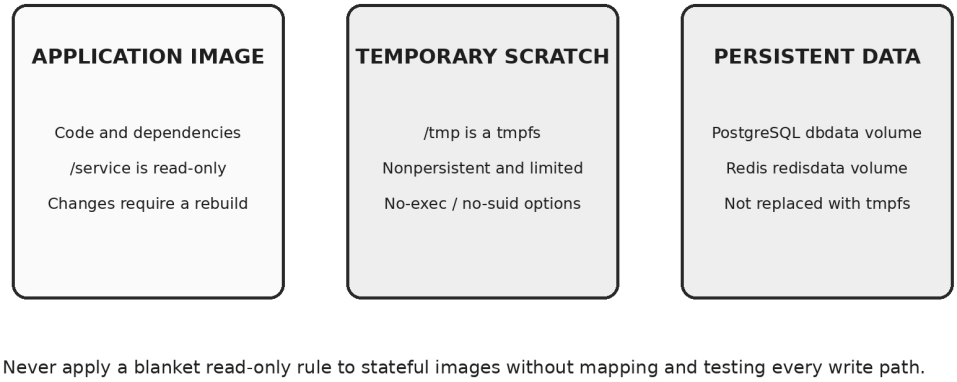


Figure 18.2 — Code is read-only; scratch space is explicitly temporary; database and Redis data remain persistent.

18.6 Set Memory and CPU Boundaries Deliberately

Limits reduce the blast radius of memory leaks and runaway work, but they are not performance optimizations by themselves. In this lab the API receives 384 MB, 1 CPU, and 128 PIDs; the publisher and worker receive 256 MB, 0.5 CPU, and 128 PIDs. These are example single-host budgets, not universal production settings. Estimate peak steady-state usage, leave space for PostgreSQL, Redis, Caddy, and the host, and adjust if measured evidence warrants it.

Observe actual usage before and after

```
docker compose -f compose.yaml -f compose.harden.yaml stats --no-stream
python tools/runtime_inspect.py
```

If a service exits with code 137 or Docker reports OOMKilled, examine its work and memory behavior before raising limits. A low limit that makes the API unstable is a failed deployment. CPU quotas constrain processing capacity but do not guarantee spare host capacity. On rootless Docker, effective cgroup limits may require cgroup v2 and systemd; inspect the daemon and verify enforcement instead of trusting the YAML alone.

Service	Example memory	CPU quota	PID ceiling
API	384 MB	1.0 CPU	128
Publisher	256 MB	0.5 CPU	128
Worker	256 MB	0.5 CPU	128
Proxy	192 MB	0.5 CPU	96
Database and Redis	Keep Chapter 16 baseline for this lab	Measure first	Harden separately

18.7 Restrict Exposed Ports and Service Networks

The existing project already has a useful perimeter: proxy publishes 127.0.0.1:8080, while db, redis, api, publisher, and worker have no host-published ports. Compose separates the edge network from an internal private network. The override preserves those definitions. The word EXPOSE in a Dockerfile is documentation—it is not a host port publication.

Verify the host boundary without listing credentials

```
docker compose -f compose.yaml -f compose.harden.yaml ps
docker compose -f compose.yaml -f compose.harden.yaml port proxy 8080
python tools/hardening_audit.py
```

Expected: the proxy maps to 127.0.0.1:8080, and the audit confirms that the database, Redis, API, publisher, and worker have no ports published to the host. On a real Linux deployment, also verify network exposure from a second machine and review Docker firewall handling. Localhost-only binding is for this controlled lab; the public deployment route is treated later.

18.8 Understand Rootless Docker Prerequisites and Limitations

Rootless Docker changes the privilege level of the Docker daemon itself. That is distinct from USER app inside an individual container. Rootless mode uses user namespaces and may require subordinate UID/GID mappings, user-level service management, and platform-specific tooling. It can also change port-forwarding behavior and storage-driver compatibility. It is a host migration decision, not a one-line addition to Compose.

Read-only inspection of daemon mode

```
docker info --format "Security options: {{json .SecurityOptions}}"
docker info --format "Cgroup driver: {{.CgroupDriver}}"
docker context ls
```

If security options include rootless, you are already using a rootless daemon context. Otherwise, do not attempt to switch modes during the checkpoint: it can change which images, containers, and volumes you see. Rootless resource constraints need a compatible cgroup configuration; confirm support before claiming memory or CPU limits are enforced. Docker Desktop architecture and remote Engine deployments also differ.

18.9 Check Security Controls Without Breaking Application Behavior

A strong policy that prevents a worker from completing jobs is not a successful release. Apply the updated Compose configuration in the original project directory, using the same .env values from Chapter 16. The first command validates the file merge, and the second recreates services as needed; it does not delete named volumes.

Apply, inspect, and exercise the hardened stack

```
docker compose -f compose.yaml -f compose.harden.yaml config --quiet
docker compose -f compose.yaml -f compose.harden.yaml up -d --build
python tools/runtime_inspect.py
docker compose -f compose.yaml -f compose.harden.yaml \
  exec -T api python tools/local_acceptance.py
```

Expected: the stack stabilizes; API readiness returns successfully; the acceptance script verifies that the proxy serves the site, missing tokens fail, viewer permissions cannot create records, operator requests can create them, and background jobs complete. The acceptance script runs inside the API container, so it can address proxy:8080 without requiring a host Python dependency. It uses the API tokens injected from the existing .env.

Failure diagnosis without disabling security

Observation	Evidence to check	Safe response
API cannot write to /tmp	tmpfs mount and mount options	Check merged Compose config; do not make entire root writable
Worker repeatedly restarts	logs, OOMKilled, resource usage	Measure memory and retries; right-size limits
403 or 401 response	token scope and request headers	Preserve API authorization model; do not bypass auth
Database has no records	Compose project name, volume identity	Confirm deployboard-ch14 and mounted dbdata
Proxy returns 502/503	API health, service DNS, startup readiness	Restore service health without publishing private ports
Permission failure in stateful service	volume ownership and image entrypoint	Do not force all database processes to arbitrary UIDs

Important: the example proxy is still a local HTTP service, and credential values are still injected as environment variables. Neither issue is concealed by the hardening audit. They are explicitly carried into Chapters 19 and 28.

18.10 Checkpoint: Produce a Working Hardened Stack

Treat this as a release decision. Your job is not to claim every checkbox is secure; it is to produce reproducible evidence that the chosen restrictions are applied and that the application’s essential behavior survives.

The seven pieces of evidence to retain

- The merged Compose configuration validates and the static hardening audit reports PASS for its intended service controls.
- Image metadata and runtime inspection prove that the three Python services run with a non-root user and without the ambient capabilities they do not need.
- The API, publisher, and worker use a read-only root filesystem; an explicit /tmp write succeeds while a /service write fails.
- The localhost-only proxy is the sole host-published service, and the database and Redis remain private.
- The authenticated CRUD, viewer/operator permissions, readiness, and worker lifecycle still pass the Chapter 16 acceptance test.
- Existing PostgreSQL records survive service recreation, and the project name and named volumes remain unchanged.
- You have recorded any limitation or failed check as an open issue, not silently relaxed the security boundary.

Final, non-destructive cleanup and rollback

```
docker compose -f compose.yaml -f compose.harden.yaml ps
# Return to the Chapter 16 runtime configuration if a control breaks a service:
docker compose -f compose.yaml up -d --force-recreate api publisher worker proxy
# Do not use: docker compose down -v
```

Rollback restores the baseline service definitions; it does not roll back database schema or delete volume contents. If a release was unsuccessful, keep the failure evidence, repair the configuration, and repeat the acceptance check. A rollback can briefly interrupt local traffic, so avoid doing it during an active operation.

What changes in the next chapter

You have reduced container privileges and narrowed writable surfaces. Chapter 19 will remove the next source of unnecessary exposure: credentials carried through runtime environment values. Do not confuse least-privilege runtime configuration with secret management; they are separate controls.

Complete verification scripts — copy into tools/

These are the complete source files required for this chapter, not pseudocode. The project already includes the Chapter 16 baseline. Save each listing under the filename shown, rebuild the Python image, and run

the audit commands from the project root. The static script never prints secrets; the runtime script inspects only selected safe fields.

File: tools/hardening_audit.py

```
"""Read-only Chapter 18 audit. Never prints credentials or a merged Compose file."""
from __future__ import annotations

import json
import subprocess
import sys

BASE = ["docker", "compose", "-f", "compose.yaml", "-f", "compose.harden.yaml"]


def capture(*args: str) -> str:
    completed = subprocess.run(
        [*args], check=True, text=True, capture_output=True
    )
    return completed.stdout.strip()


def verdict(label: str, okay: bool) -> bool:
    print(f"{'PASS' if okay else 'FAIL'}  {label}")
    return okay


def main() -> int:
    good = True
    try:
        # docker compose config may include interpolated secrets. Parse, NEVER print.
        cfg = json.loads(capture(*BASE, "config", "--format", "json"))
    except (subprocess.CalledProcessError, FileNotFoundError, ValueError) as exc:
        print("ERROR: Compose config was unavailable; check .env and Docker tools.")
        return 2

    services = cfg.get("services", {})
    for name in ("api", "publisher", "worker"):
        svc = services.get(name, {})
        good &= verdict(f"{name}: read-only root filesystem", svc.get("read_only") is True)
        good &= verdict(f"{name}: all capabilities dropped",
                        "ALL" in [str(x).upper() for x in svc.get("cap_drop", [])])
        good &= verdict(f"{name}: no-new-privileges",
                        any(str(x).startswith("no-new-privileges") for x in svc.get("security_opt", [])))
        good &= verdict(f"{name}: scratch tmpfs",
                        any(str(x).startswith("/tmp:") for x in svc.get("tmpfs", [])))
        good &= verdict(f"{name}: memory and process limits",
                        bool(svc.get("mem_limit")) and int(svc.get("pids_limit", 0)) > 0)

    for name in ("db", "redis", "api", "publisher", "worker"):
        svc = services.get(name, {})
        good &= verdict(f"{name}: no host-published ports", not svc.get("ports"))
    ports = services.get("proxy", {}).get("ports", [])
    good &= verdict("proxy: expected local-only published port",
                    len(ports) == 1 and all(
                        (p.get("host_ip") == "127.0.0.1") if isinstance(p, dict)
                        else str(p).startswith("127.0.0.1:") for p in ports))
    print("Review stopped here: this is static policy verification, not a runtime proof.")
    return 0 if good else 1


if __name__ == "__main__":
    sys.exit(main())
```

File: tools/runtime_inspect.py

```
"""Report selected runtime controls without showing environment variables."""
from __future__ import annotations

import json
import subprocess
```

```
import sys

BASE = ["docker", "compose", "-f", "compose.yaml", "-f", "compose.harden.yaml"]

def output(cmd: list[str]) -> str:
    return subprocess.run(cmd, check=True, capture_output=True, text=True).stdout.strip()

def main() -> int:
    okay = True
    for service in ("api", "publisher", "worker", "proxy"):
        try:
            cid = output([*BASE, "ps", "-q", service])
            if not cid:
                print(f"NOT RUNNING {service}")
                okay = False
                continue
            item = json.loads(output(["docker", "inspect", cid]))[0]
        except (OSError, subprocess.CalledProcessError, ValueError, IndexError):
            print(f"INSPECTION FAILED {service}")
            okay = False
            continue
        hc = item["HostConfig"]
        user = item["Config"].get("User") or "(image default)"
        caps = [str(c).upper() for c in hc.get("CapDrop") or []]
        secure = any(str(x).startswith("no-new-privileges")
                     for x in hc.get("SecurityOpt") or [])
        ro = bool(hc.get("ReadonlyRootfs"))
        print(f"{service}: state={item['State']['Status']}; user={user}; "
              f"read_only={ro}; drop_all={'ALL' in caps}; "
              f"no_new_privs={secure}; memory_bytes={hc.get('Memory', 0)}; "
              f"pids_limit={hc.get('PidsLimit', 0)}")
        if service in ("api", "publisher", "worker"):
            okay &= ro and ("ALL" in caps) and secure and bool(user)
    return 0 if okay else 1

if __name__ == "__main__":
    sys.exit(main())
```

File: tools/prove_writes.py

```
"""Verify writable scratch space AND read-only application code path."""
from __future__ import annotations

import errno
import sys
from pathlib import Path

scratch = Path('/tmp/ch18-proof')
code_target = Path('/service/app/ch18-should-fail')

try:
    scratch.write_text('scratch-ok', encoding='utf-8')
    good = scratch.read_text(encoding='utf-8') == 'scratch-ok'
    scratch.unlink()
    print('PASS: /tmp write/read/remove' if good else 'FAIL: scratch mismatch')
except OSError as exc:
    print(f'FAIL: /tmp is not usable ({exc.__class__.__name__})')
    sys.exit(1)

try:
    code_target.write_text('not-allowed', encoding='utf-8')
except OSError as exc:
    if exc.errno == errno.EROFS:
        print('PASS: /service/app is read-only')
        sys.exit(0 if good else 1)
    print(f'FAIL: write rejected for another reason ({exc.__class__.__name__})')
    sys.exit(1)
else:
```

```
code_target.unlink(missing_ok=True)
print('FAIL: application code was writable')
sys.exit(1)
```

Security release evidence sheet — complete after running the stack

For your final local-release record, fill in actual values from your own host. Keep tokens and passwords out of these notes. A PASS is meaningful only when backed by observed output—not by the presence of an option in YAML.

Evidence item	Observed result / record
Docker / Compose versions	_____
API, worker, publisher runtime UID	_____
Read-only + /tmp proof	_____
Capabilities / NoNewPrivs inspection	_____
Proxy bind address and port	_____
Authenticated CRUD and job lifecycle	_____
Persistence and project volume identity	_____
Unresolved issue and rollback decision	_____

Release disposition: ACCEPT only if each applicable check is supported by evidence, records remain intact, and the full application acceptance test succeeds. Otherwise REJECT, document the failure, and return to the known-good Chapter 16 service configuration without removing volumes.

Chapter 19 — Handle Secrets and Configuration Without Leaks

A teammate runs a routine diagnostics command and discovers an API token in the container environment. The application works, but its deployment model has made a credential too easy to disclose. This chapter fixes that specific weakness without replacing your PostgreSQL data, abandoning the least-privilege controls from Chapter 18, or changing DeployBoard's authenticated HTTP contract.

Delivery decision: Move credentials out of Dockerfile ARG/ENV and Compose environment values. Use narrowly granted file mounts for running Linux containers, BuildKit mounts for build-time authentication, and deliberate rotation and verification procedures.

Working baseline: The Chapter 18 Compose project is named deployboard-ch14. It contains api, publisher, worker, db, redis, proxy, and an optional migrate service. Keep that project identity so named volumes remain associated with the existing deployment.

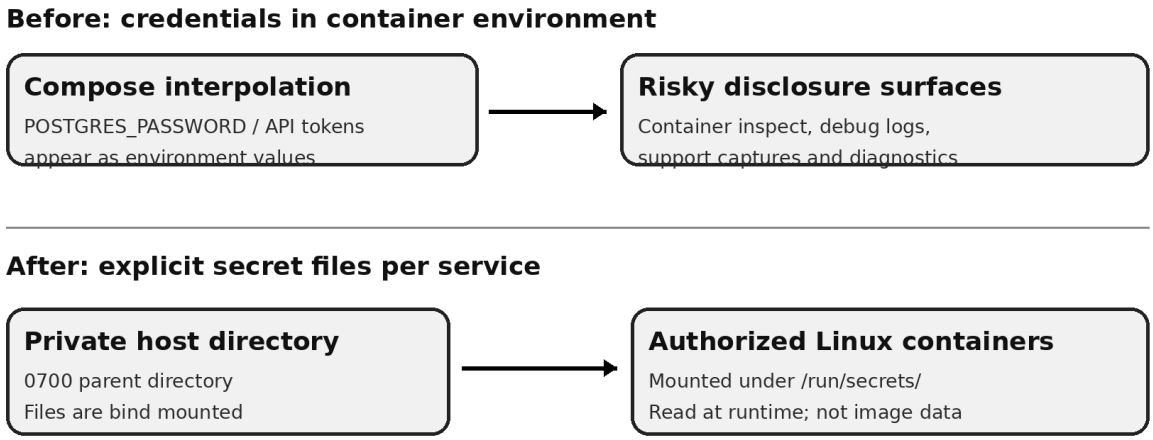


Figure 19.1 — Credential handling before and after the Compose migration.

19.1 Separate Public Configuration from Secrets

Configuration describes how the application should behave. A secret authorizes access to a protected resource. Mixing the two makes diagnostics, support bundles, and build metadata unnecessarily sensitive.

Setting or material	Classification	Preferred handling
DEPLOYBOARD_LABEL, DB_HOST	Ordinary configuration	Compose environment
POSTGRES password	Database credential	File secret for db and Python services
Viewer / operator tokens	Authentication credentials	File secrets for api only
REDIS_HOST and feature toggles	Ordinary configuration	Compose environment
Private registry access token	Build or CI credential	BuildKit secret / protected CI credential
TLS private key	High-impact secret	Restricted host path or managed secret store

A secret file is not automatically secure simply because its filename contains “secret.” Protect the source path, control which services receive the mount, and avoid printing the content anywhere in the verification workflow.

19.2 Explain Why Dockerfile ARG and ENV Are Not Secret Stores

Dockerfile ARG and ENV are configuration mechanisms. Values can appear in build metadata, image configuration, build history, logs, cache artifacts, or other retained records depending on how they are used. Never assume that a later RUN rm or an additional layer erases an earlier disclosure.

Unsafe pattern to recognize (not a lab command)

```
# Illustrative ANTI-PATTERN - do not use with real credentials.
ARG PRIVATE_REGISTRY_TOKEN
ENV PRIVATE_REGISTRY_TOKEN=$PRIVATE_REGISTRY_TOKEN
RUN echo "Do not print tokens in build logs"
```

Use ARG for non-secret build options such as target platform, and ENV for harmless runtime defaults. For actual credentials required only during a build, use BuildKit secrets. For credentials needed after container startup, use runtime secret files or an external secret manager.

19.3 Use BuildKit Secret Mounts During Image Builds

BuildKit mounts a secret for a single RUN instruction. The mount is not copied into the output image unless your command deliberately copies or prints it. The following safe proof uses a disposable string; it is not a private registry login.

Complete file: Dockerfile.secret-check

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim
RUN --mount=type=secret,id=build_proof,required=true \
    python -c "from pathlib import Path; assert Path('/run/secrets/build_proof').stat().st_size > 0"
CMD ["python", "-c", "print('BuildKit secret used without persisting its file')"]
```

Portable commands, run from project root

```
python -c "from pathlib import Path; Path('private/build_proof.txt').write_text('lab-only-build-input\n')"
docker buildx build --secret type=file,id=build_proof,src=private/build_proof.txt --load -t deployboard:build-secret-proof -f Dockerfile.secret-check .
docker run --rm deployboard:build-secret-proof
docker run --rm --entrypoint python deployboard:build-secret-proof -c "from pathlib import Path; assert not Path('/run/secrets/build_proof').exists(); print('PASS: no runtime build secret file')"
```

Expected result: the build succeeds only when build_proof is supplied; the test container reports that no mount exists at runtime. Dockerfile instructions still appear in metadata, so avoid embedding secret values in command text.

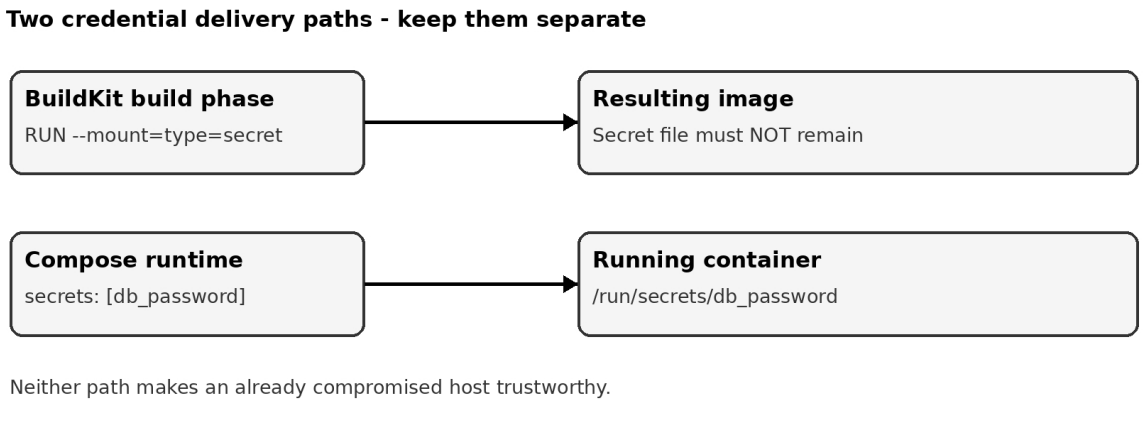


Figure 19.2 — Build-time secret mounts and Compose runtime mounts serve different purposes.

19.4 Mount Compose Secrets as Files in Linux Containers

Compose file secrets are declared at the top level and granted to individual services. In this book's Linux-container stack, Docker exposes them under /run/secrets/. PostgreSQL's official image accepts POSTGRES_PASSWORD_FILE; the Python services will receive only a *_FILE setting and read the mounted file themselves.

Compatibility requirement: Keep the existing compose.harden.yaml from Chapter 18. Replace the earlier compose.yaml with the complete Chapter 19 version below; do not try to "remove" old sensitive environment keys using null entries in an additive override, because environment merging can be misleading.

Complete file: compose.yaml (replaces the Chapter 18 baseline)

```
# Chapter 19. Keep project name to reuse Chapter 14-18 volumes.
# Secrets are local read-only mounts, not encrypted Swarm secrets.
name: deployboard-ch14

x-app-env: &app_env
  DB_HOST: db
  DB_NAME: deployboard
  DB_USER: deployboard
  DB_PASSWORD_FILE: /run/secrets/db_password
  REDIS_HOST: redis

services:
  db:
    image: postgres:17
    environment:
      POSTGRES_USER: deployboard
      POSTGRES_DB: deployboard
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets: [db_password]
    volumes:
      - dbdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
      interval: 5s
      timeout: 3s
      retries: 10
      start_period: 15s
    restart: unless-stopped
    networks: [private]

  redis:
    image: redis:7.4.11-bookworm
    command: ["redis-server", "--appendonly", "yes", "--appendfsync", "everysec"]
```

```
    volumes: ["redisdata:/data"]
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 10
    restart: unless-stopped
    networks: [private]

  api:
    build: .
    image: deployboard:ch19
    environment:
      <<: *app_env
      DEPLOYBOARD_VIEWER_TOKEN_FILE: /run/secrets/viewer_token
      DEPLOYBOARD_OPERATOR_TOKEN_FILE: /run/secrets/operator_token
    secrets: [db_password, viewer_token, operator_token]
    depends_on:
      db: {condition: service_healthy}
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request;
urllib.request.urlopen('http://127.0.0.1:8000/ready', timeout=3)"]
      interval: 10s
      timeout: 4s
      retries: 3
      start_period: 20s
    restart: unless-stopped
    networks: [edge, private]

  publisher:
    image: deployboard:ch19
    command: ["python", "-m", "app.publisher"]
    environment: *app_env
    secrets: [db_password]
    depends_on:
      db: {condition: service_healthy}
      redis: {condition: service_healthy}
    restart: unless-stopped
    networks: [private]

  worker:
    image: deployboard:ch19
    command: ["python", "-m", "app.worker"]
    environment:
      <<: *app_env
      DEPLOYBOARD_FAIL_FIRST: ${DEPLOYBOARD_FAIL_FIRST:-0}
    secrets: [db_password]
    depends_on:
```

```
      db: {condition: service_healthy}
      redis: {condition: service_healthy}
    stop_grace_period: 15s
    restart: unless-stopped
    networks: [private]

  proxy:
    image: caddy:2.11.4-alpine
    ports: ["127.0.0.1:8080:8080"]
    volumes:
      - ./proxy/Caddyfile:/etc/caddy/Caddyfile:ro
      - ./site:/srv:ro
    depends_on:
      api: {condition: service_healthy}
    restart: unless-stopped
    networks: [edge]

  migrate:
    image: deployboard:ch19
    profiles: [tools]
    environment: *app_env
    secrets: [db_password]
    depends_on:
      db: {condition: service_healthy}
    command: ["alembic", "upgrade", "head"]
    networks: [private]

volumes:
  dbdata:
  redisdata:

networks:
  edge:
    driver: bridge
  private:
    driver: bridge
    internal: true

secrets:
  db_password:
    file: ./private/db_password.txt
  viewer_token:
    file: ./private/viewer_token.txt
  operator_token:
    file: ./private/operator_token.txt
```

Only the proxy publishes a port, and it binds to 127.0.0.1:8080. PostgreSQL and Redis stay on the private network. The db_password secret is available to db, api, publisher, worker, and migrate; the viewer and operator token files are granted only to api.

Compose file sources are implemented with bind mounts; unlike some Swarm or other secret sources, file-backed mounts do not give you reliable container UID/GID remapping. We validate readability by the non-root application instead of assuming mode: or uid: in YAML overrides host file permissions.

19.5 Adapt Applications to Read File-Mounted Credentials

The existing application and queue logic already accept a Settings object. We can switch credential delivery without rewriting CRUD routes, the database schema, or worker idempotency. The helper rejects a credential that appears both as a plaintext environment variable and as a *_FILE source, avoiding ambiguous precedence.

Complete replacement: app/settings.py

```
"""Credential file support for DeployBoard, retaining Chapter 13 Settings API."""
from dataclasses import dataclass
from pathlib import Path
import os
import secrets


def _credential(name: str) -> str:
    """Prefer explicit /run/secrets files; reject ambiguous dual sources."""
    filepath = os.getenv(name + "_FILE", "")
    literal = os.getenv(name, "")
    if filepath and literal:
        raise ValueError(f"Configure either {name} or {name}_FILE, not both")
    if filepath:
        # Trim newline only, not spaces that could be part of a password.
        return Path(filepath).read_text(encoding="utf-8").rstrip("\r\n")
    return literal


@dataclass(frozen=True)
class Settings:
    viewer_token: str = ""
    operator_token: str = ""
```

```
  service_label: str = "DeployBoard"
  db_host: str = "db"
  db_port: int = 5432
  db_name: str = "deployboard"
  db_user: str = "deployboard"
  db_password: str = ""

  @classmethod
  def from_env(cls) -> "Settings":
      return cls(
          viewer_token=_credential("DEPLOYBOARD_VIEWER_TOKEN"),
          operator_token=_credential("DEPLOYBOARD_OPERATOR_TOKEN"),
          service_label=os.getenv("DEPLOYBOARD_LABEL", "DeployBoard"),
          db_host=os.getenv("DB_HOST", "db"),
          db_port=int(os.getenv("DB_PORT", "5432")),
          db_name=os.getenv("DB_NAME", "deployboard"),
          db_user=os.getenv("DB_USER", "deployboard"),
          db_password=_credential("DB_PASSWORD"),
      )

  @property
  def valid_tokens(self) -> bool:
      return (len(self.viewer_token) >= 20
              and len(self.operator_token) >= 20
              and not self.viewer_token.startswith("REPLACE_")
              and not self.operator_token.startswith("REPLACE_")
              and not secrets.compare_digest(self.viewer_token,
                                             self.operator_token))

  @property
  def valid_database(self) -> bool:
      return bool(self.db_host and self.db_name and self.db_user
                  and self.db_password
                  and not self.db_password.startswith("REPLACE_"))
```

Existing tool compatibility: Update the prior lab tools so they use the settings adapter instead of reading API token values directly from environment variables. The bundled Alembic env.py already uses this adapter; all application routes and schema objects remain unchanged.

Precise migration edits for Chapter 15-16 utility scripts

```
  # For tools/local_acceptance.py (replace the existing token assignments):
  from app.settings import Settings
  AUTH = Settings.from_env()
  VIEWER, OPERATOR = AUTH.viewer_token, AUTH.operator_token

  # For tools/check_jobs.py (inside main, replace token assignments):
```

```
auth = Settings.from_env()  # also import Settings at module top
operator, viewer = auth.operator_token, auth.viewer_token

# For tools/verify_record.py (replace Authorization token source):
from app.settings import Settings
# ... in your Request headers: "Bearer " + Settings.from_env().viewer_token

# For tools/smoke.py (replace token assignments):
auth = Settings.from_env()  # also import Settings at module top
viewer, operator = auth.viewer_token, auth.operator_token
```

Inside a running API container, confirm file access without displaying values:

```
docker compose -f compose.yaml -f compose.harden.yaml exec -T api python -c "from
app.settings import Settings; s=Settings.from_env(); assert s.valid_tokens and s.valid_database;
print('PASS: file-backed credentials loaded')"
```

Failure conditions are intentional: an absent secret path raises an error, and conflicting environment and file settings fail immediately. This is safer than quietly starting the service with an empty or unexpected credential.

19.6 Protect Local Secret Files and Ignore Rules

For a NEW lab, the preparation utility generates random credentials. When upgrading an EXISTING volume-backed PostgreSQL installation, it reuses the three values in the existing .env so the app does not lose access to its database. Never publish or copy that .env to a repository.

Complete file: tools/prepare_ch19_secrets.py

```
"""Prepare file-backed secrets safely; preserve existing PostgreSQL data.

Run from the Compose project root. Does not print credential values.
For an existing database, reuse its EXACT current password from .env.
"""
from pathlib import Path
import os
import secrets

DIRECTORY = Path("private")
MAPPING = {
    "POSTGRES_PASSWORD": "db_password.txt",
    "DEPLOYBOARD_VIEWER_TOKEN": "viewer_token.txt",
    "DEPLOYBOARD_OPERATOR_TOKEN": "operator_token.txt",
}
```

```
def previous_values() -> dict[str, str]:
    source = Path(".env")
    if not source.exists():
        return {}
    values = {}
    for raw in source.read_text(encoding="utf-8").splitlines():
        raw = raw.strip()
        if not raw or raw.startswith("#") or "=" not in raw:
            continue
        key, value = raw.split("=", 1)
        values[key.strip()] = value.strip().strip("'\"")
    return values


def main() -> None:
    DIRECTORY.mkdir(mode=0o700, exist_ok=True)
    if DIRECTORY.is_symlink():
        raise SystemExit("Refusing a symlinked private directory")
    if os.name == "posix":
        DIRECTORY.chmod(0o700)
    existing = previous_values()
    targets = [DIRECTORY / n for n in MAPPING.values()]
    if any(t.exists() or t.is_symlink() for t in targets):
        raise SystemExit("Secret files already exist; refusing to overwrite")
    for variable, name in MAPPING.items():
        value = existing.get(variable) or secrets.token_urlsafe(32)
        if value.startswith("REPLACE_"):
            raise SystemExit(f"Replace placeholder {variable} before continuing")
        target = DIRECTORY / name
        with target.open("x", encoding="utf-8") as handle:
            handle.write(value + "\n")
        if os.name == "posix":
            # The parent directory (0700) protects the source path on the host.
            # The mounted file is readable by a non-root container service.
            target.chmod(0o444)
    print("Prepared 3 secret files; values were not printed")
    print("Preserve .env until the Compose migration and acceptance check pass")

if __name__ == "__main__":
    main()
```

Complete file: .gitignore

```
.env
private/
*.pyc
__pycache__/
.pytest_cache/
```

```
.git/
```

Complete file: .dockerignore

```
.git/
.env
.env.*
!.env.example
private/
__pycache__/
*.pyc
.pytest_cache/
tests/
*.zip
*.db
site/
proxy/
```

Preparation and validation - do not display file contents

```
python tools/prepare_ch19_secrets.py
python -c "from pathlib import Path; root=Path('private'); assert all((root/p).is_file() for p in ['db_password.txt','viewer_token.txt','operator_token.txt']); print('PASS: secret files present')"
docker compose -f compose.yaml -f compose.harden.yaml config --quiet
```

POSIX lab layout: directory private/ is mode 0700; mounted files are read-only 0444 so the Chapter 18 non-root service can read them. The protected parent prevents other ordinary host users from traversing the path, but administrators and Docker daemon operators are still powerful trust principals. On Windows and macOS, verify actual filesystem sharing and ACL behavior; chmod alone is not a cross-platform host security guarantee.

19.7 Configure CI Credentials with Least Privilege

A CI job that tests an untrusted pull request must not automatically receive release credentials. Separate build/test permissions from release/publish permissions; limit scope to the specific registry and environment; use short-lived identity-based credentials where supported. Protected release jobs should require a trusted branch or an explicit approval policy.

CI policy fragment - workflow implementation belongs to Chapter 26

```
# Minimal permissions principle (illustrative GitHub Actions fragment)
permissions:
  contents: read
jobs:
```

```
tests:
  permissions:
    contents: read
publish:
  if: github.event_name == 'push' && github.ref == 'refs/heads/main'
  permissions:
    contents: read
    packages: write
```

Never store credentials in the repository, the Docker build context, test artifacts, or printed command history. Do not paste secrets into issue trackers or AI prompts. GitHub or another CI provider is optional for the local workflows in this chapter.

19.8 Rotate Credentials and Restart Dependent Services

Rotation is a coordinated change: create the new credential, roll it out to the correct service, update authorized clients, verify authentication, and retire the previous value. This script replaces one API token file atomically on the host. Because bind mounts can refer to an earlier file inode, recreate the API service afterward.

Complete file: tools/rotate_ch19_token.py

```
"""Rotate one local API token without exposing its value on stdout.

After running, RECREATE api so Compose rebinds the replacement file.
Old API clients must switch credentials at the deployment window.
"""
import argparse
from pathlib import Path
import os
import secrets

parser = argparse.ArgumentParser()
parser.add_argument("role", choices=["viewer", "operator"])
args = parser.parse_args()
root = Path("private")
name = root / (args.role + "_token.txt")
if not name.is_file() or name.is_symlink():
    raise SystemExit("Run the secret preparation step first")
new = name.with_suffix(".next")
if new.exists():
    raise SystemExit("Remove stale .next only after inspecting it")
with new.open("x", encoding="utf-8") as out:
```

```
    out.write(secrets.token_urlsafe(32) + "\n")
if os.name == "posix":
    new.chmod(0o444)
os.replace(new, name)
print("Rotated token file. Recreate the API container; coordinate client updates")
```

Rotate only after arranging new credentials for API clients

```
python tools/rotate_ch19_token.py operator
docker compose -f compose.yaml -f compose.harden.yaml up -d --no-deps --force-recreate api
docker compose -f compose.yaml -f compose.harden.yaml exec -T api python -c "from
app.settings import Settings; assert Settings.from_env().valid_tokens; print('PASS: current file
tokens loaded')"
```

Do not rotate an existing PostgreSQL password merely by editing db_password.txt. PostgreSQL initializes database credentials on first volume creation; a preexisting database user password must be changed with an authenticated SQL operation, carefully coordinated with the mounted file and dependent services. Back up first, then verify both migration access and runtime connections.

19.9 Avoid Leaking Tokens Through Logs and Debug Output

Do not print bearer tokens, database URLs containing passwords, full request headers, private .env files, or secret-file contents. Use correlation IDs and sanitized names when troubleshooting. In particular, do not paste unredacted docker inspect output into tickets or screenshots when secrets may still be present in older containers.

Evidence without disclosure: Report “token file exists,” “API authenticates,” “container environment lacks sensitive value keys,” and “BuildKit secret was not persisted.” Never print or record the token just to show that it exists.

Safe verification sequence

```
docker compose -f compose.yaml -f compose.harden.yaml config --quiet
python tools/secret_config_audit.py
python -m pytest -q tests/test_secret_files.py
```

The secret_config_audit utility reads Compose’s JSON configuration and checks key names and service grants without printing credential values. Its

full implementation appears below. It requires a working Docker Compose CLI; this is a structural check, not a substitute for runtime inspection.

Complete file: tools/secret_config_audit.py

```
"""Local source and Compose structural guard; never print secret values."""
import json
import subprocess
from pathlib import Path

root = Path(__file__).resolve().parents[1]
config = subprocess.run(
    ["docker", "compose", "-f", "compose.yaml", "-f", "compose.harden.yaml",
     "config", "--format", "json"],
    cwd=root, check=True, text=True, capture_output=True,
)
model = json.loads(config.stdout)
expected = {"api", "publisher", "worker", "db", "migrate"}
for service in expected:
    assert service in model["services"], service
for key, entry in model["services"].items():
    env = entry.get("environment") or {}
    for forbidden in ["POSTGRES_PASSWORD", "DB_PASSWORD",
                      "DEPLOYBOARD_VIEWER_TOKEN",
"DEPLOYBOARD_OPERATOR_TOKEN"]:
        assert forbidden not in env, f"{key}: forbidden environment key {forbidden}"
assert set(model["services"]["api"]["secrets"][i]["source"]
           for i in range(len(model["services"]["api"]["secrets"]))) == {
               "db_password", "viewer_token", "operator_token"}
assert "ports" not in model["services"]["db"]
assert "ports" not in model["services"]["redis"]
print("PASS: secret references, no credential-valued environment keys, private data ports")
```

19.10 Compare Compose File-Mounted Secrets with Swarm-Managed Secrets

Dimension	Compose file secrets	Swarm-managed secrets
Source of truth	Host file path	Docker Swarm secret object
Storage protection	Host filesystem controls; not automatic encryption	Encrypted in Swarm Raft log; access granted to tasks
Container delivery	File bind mount for Linux containers	Managed secret mount in service task
Access and rotation	Host file and service recreation; explicit runbook	New secret version and service update
Use in this book	Single-host Compose lab	Optional future orchestration topic

Both systems require effective host security, privileged-operator controls, and deliberate secret rotation. “Available under /run/secrets” describes the container interface, not a promise of identical security properties.

19.11 Limit Host Secret-File Access and Plan Secure Alternatives

Store local secret sources outside images and Git. Keep private/ out of build context, restrict its parent directory, use restrictive host ACLs, and minimize users in the docker group. Back up secrets separately from the public source tree only when recovery requires it; encrypt backups and document authorized access.

For higher-risk production environments, prefer a dedicated secret-management system capable of centralized auditing, identity-based access, rotation, and short-lived credentials. A Compose file mount is an improvement over embedding secrets in images; it is not a substitute for host compromise protection or organizational key management.

19.12 Checkpoint: Verify No Secret Is Embedded in an Image

Your release candidate must pass an evidence-based inspection, not a visual guess. Work from the same project folder and avoid recreating database volumes. The focused tests below run without exposing any real credential values.

Complete file: tests/test_secret_files.py

```
"""Focused tests for file-backed settings; no real tokens are included."""
from pathlib import Path
import pytest
from app.settings import Settings


def test_file_credentials_override_absent_environment(tmp_path, monkeypatch):
    db = tmp_path / "db"
    db.write_text("test-database-secret\n", encoding="utf-8")
    viewer = tmp_path / "viewer"
    viewer.write_text("v" * 32 + "\n", encoding="utf-8")
    operator = tmp_path / "operator"
    operator.write_text("o" * 32 + "\n", encoding="utf-8")
```

```
    for k in ("DB_PASSWORD", "DEPLOYBOARD_VIEWER_TOKEN",
"DEPLOYBOARD_OPERATOR_TOKEN"):
        monkeypatch.delenv(k, raising=False)
    monkeypatch.setenv("DB_PASSWORD_FILE", str(db))
    monkeypatch.setenv("DEPLOYBOARD_VIEWER_TOKEN_FILE", str(viewer))
    monkeypatch.setenv("DEPLOYBOARD_OPERATOR_TOKEN_FILE", str(operator))
    s = Settings.from_env()
    assert s.valid_tokens and s.valid_database
    assert s.db_password == "test-database-secret"


def test_reject_both_sources(tmp_path, monkeypatch):
    file = tmp_path / "key"
    file.write_text("secret", encoding="utf-8")
    monkeypatch.setenv("DB_PASSWORD_FILE", str(file))
    monkeypatch.setenv("DB_PASSWORD", "legacy-env-value")
    with pytest.raises(ValueError, match="either"):
        Settings.from_env()


def test_missing_file_fails_closed(tmp_path, monkeypatch):
    monkeypatch.setenv("DB_PASSWORD_FILE", str(tmp_path / "absent"))
    monkeypatch.delenv("DB_PASSWORD", raising=False)
    with pytest.raises(FileNotFoundError):
        Settings.from_env()
```

Acceptance sequence (run after the BuildKit proof from 19.3)

```
python -m pytest -q tests/test_secret_files.py tests/test_api.py
docker compose -f compose.yaml -f compose.harden.yaml config --quiet
python tools/secret_config_audit.py
docker compose -f compose.yaml -f compose.harden.yaml up -d --build
docker compose -f compose.yaml -f compose.harden.yaml exec -T api python
tools/local_acceptance.py
docker image inspect deployboard:ch19 --format '{{json .Config.Env}}'
docker run --rm --entrypoint python deployboard:build-secret-proof -c "from pathlib import Path;
assert not Path('/run/secrets/build_proof').exists(); print('PASS')"
```

Inspect docker image inspect results for harmless environment settings only. The command prints image ENV metadata, not the mounted runtime secrets; avoid using it if you previously built an insecure image with real credentials. Never run docker compose down -v on a project whose PostgreSQL records you intend to preserve.

Release evidence	Pass condition
Source-control hygiene	.gitignore and .dockerignore exclude private/
Compose model	No raw credential environment keys in services

Release evidence	Pass condition
Service grants	Only API receives API tokens; all relevant DB clients receive DB secret
Functional compatibility	Existing authenticated CRUD and background job still work
Image inspection	No secret value stored in image ENV or output filesystem
Rotation runbook	New token deployed and client access reverified
Rollback readiness	Database volume untouched; prior protected configuration retained

Outcome: DeployBoard still serves the same authenticated API and PostgreSQL-backed background jobs, but credentials are now delivered through files to the services that need them. Chapter 20 extends the evidence trail from secret handling to image vulnerabilities, SBOMs, and provenance.

Chapter 20 — Inspect Image Risks and Build Supply Chain Evidence

A release can pass every application test and still carry a vulnerable library, an unreviewed base image, or a build that nobody can trace. Your next task is not to claim DeployBoard is secure. It is to make its origin, contents, risks, and approval decision inspectable.

You will review the Chapter 19 image without changing its authenticated routes, PostgreSQL data, Redis worker, or secret-file contracts. The required path is local and free. Publishing attestations to your own registry is a clearly marked optional extension; registry sign-in and network access are only needed there.

Your acceptance evidence

- An image built from the preserved DeployBoard source and inspected by identifier.
- A machine-readable vulnerability report and SBOM generated from that image.
- A review summary that never interprets an empty report as automatic approval.
- A documented release decision tied to a digest if—and only if—you publish to a registry.

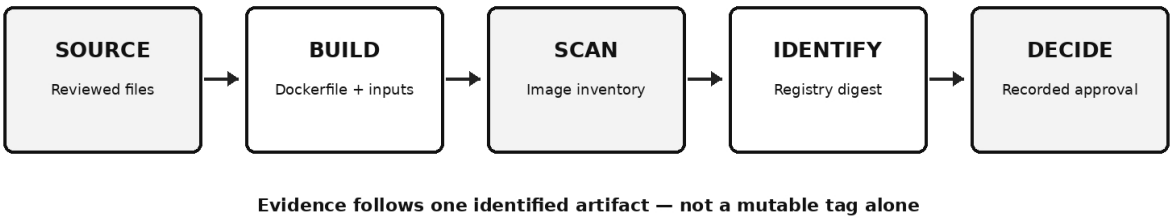


Figure 20.1 — An image becomes a candidate for release only after identifiable evidence and a deliberate review.

20.1 Establish Trusted Base-Image Selection Rules

The first trust decision occurs before your application code is copied: Docker must obtain a base image. Chapter 19 uses python:3.13-slim. That is a sensible compact image family, but its tag can move when upstream maintainers publish an update. “Official” is not synonymous with patched, and a pinned digest is not a promise that the pinned version will remain safe.

Review a candidate before replacing the base

Commands 20.1 — Inspect published manifests and the local application platform

```
docker buildx imagetools inspect python:3.13-slim
docker image inspect deployboard:ch19 --format "{{.Os}}/{{.Architecture}}"
```

In the manifest output, record the top-level index digest and the applicable platform. An index digest identifies a multi-platform index; a platform-specific manifest digest identifies the corresponding platform image. Do not confuse either with a locally printed image ID.

Question	Good release practice
Who maintains it?	Prefer identifiable maintainers and an actively updated image.
Is this platform supported?	Confirm linux/amd64 or linux/arm64 as appropriate.
Which digest was reviewed?	Record the exact digest with its platform and review date.
Can you update it?	Plan scheduled rebuilds; pinning alone does not patch vulnerabilities.

Before release, replace the mutable FROM reference with the real digest you inspected, using FROM python:3.13-slim@sha256:<actual_digest>. The angle-bracket text is a placeholder, not a runnable digest. Verify the architecture and rebuild after choosing one. Keep the unmodified Chapter 19 Dockerfile in your working source until you have a valid digest.

20.2 Understand Common Vulnerability Reports and Severity

A scanner compares discovered packages against advisory data. Its output is a set of potential matches, not an exploit demonstration. A HIGH label can have low practical impact on an unreachable code path; a MEDIUM issue

exposed to untrusted traffic can still merit urgent work. Evaluate exploitability, affected package, available fix, and operational exposure together.

Term	What it tells you
CVE	A public identifier for a reported vulnerability.
CVSS / severity	A technical severity estimate; not an automatic release decision.
FixedVersion	The version an advisory says contains a fix, when known.
CISA KEV / exploit evidence	Additional indication that an issue may be exploited in practice.
False positive / not affected	Requires traceable supporting justification, not silent deletion.

Scan results change as advisory databases improve. Record both the scanned artifact and when the scan was produced. Do not copy a reassuring number of findings from this book into your own release evidence.

20.3 Run an Image Vulnerability Scan with a Documented Tool

Use Trivy, an image and software-inventory scanner. Install it using the official Trivy installation instructions for your system, then confirm trivy --version. For the core lab you need only Docker, Trivy, Python 3, and the source files already supplied in Chapters 13–19. Run commands from the DeployBoard project root.

Commands 20.2 — Build and scan the local review image

```
docker buildx version
trivy --version
docker buildx build --load -t deployboard:ch20-review .
trivy image --scanners vuln --format json --output evidence/scan.json deployboard:ch20-review
```

For the first run Trivy may download advisory databases. Do not call a network failure “zero findings.” If scanning fails, stop and repair the environment. You can view a concise table without changing the report with the next command.

Command 20.3 — Inspect high-priority findings

```
trivy image --scanners vuln --severity HIGH,CRITICAL deployboard:ch20-review
```

The generated evidence/scan.json is specific to your installed packages and scanner database. The tag ch20-review is convenient for the exercise, but it

is not an immutable registry reference. Never claim production approval on this tag alone.

20.4 Decide What to Patch, Accept, or Investigate

Your job is triage, not chasing a zero. A fixable vulnerable Python dependency may be repaired by updating a tested requirement and rebuilding; an OS package issue may require a refreshed base image. An advisory with no fix needs documented exposure assessment, mitigation, ownership, and a recheck date. Every exception requires a named reviewer.

A conservative machine-assisted first pass

The following standard-library script turns Trivy JSON into a short review record. HIGH or CRITICAL findings trigger a hold. Absence of such findings still yields MANUAL_REVIEW_REQUIRED; this is intentional. The script does not decide whether the software is safe.

File: review/review_image.py — complete listing

```
#!/usr/bin/env python3
"""Summarize a Trivy JSON report; never auto-approve a release."""
import argparse
import hashlib
import json
from pathlib import Path

SEVERITIES = ("CRITICAL", "HIGH", "MEDIUM", "LOW", "UNKNOWN")


def summarize(data):
    if not isinstance(data, dict) or not isinstance(data.get("Results"), list):
        raise ValueError("Expected Trivy JSON with a Results array")
    if not data.get("ArtifactName"):
        raise ValueError("Missing scanned ArtifactName; do not infer success")
    counts = {key: 0 for key in SEVERITIES}
    priority = []
    for result in data["Results"]:
        for item in result.get("Vulnerabilities") or []:
            severity = str(item.get("Severity", "UNKNOWN")).upper()
            severity = severity if severity in counts else "UNKNOWN"
            counts[severity] += 1
            if severity in ("CRITICAL", "HIGH"):
                priority.append({
                    "id": item.get("VulnerabilityID", "unknown"),
                    "package": item.get("PkgName", "unknown"),
                    "installed": item.get("InstalledVersion", "unknown"),
                    "fixed": item.get("FixedVersion") or "not listed",
                    "severity": severity,
                })
    blocked = counts["CRITICAL"] + counts["HIGH"] > 0
    return {
        "artifact": data["ArtifactName"],
        "counts": counts,
        "disposition": "HOLD_FOR_TRIAGE" if blocked else "MANUAL_REVIEW_REQUIRED",
        "priority_findings": priority[:30],
```

```
        "note": "A scan is a point-in-time signal, not proof of safety.",
    }


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("scan", type=Path)
    parser.add_argument("output", type=Path)
    args = parser.parse_args()
    raw = args.scan.read_bytes()
    report = summarize(json.loads(raw))
    report["scan_sha256"] = hashlib.sha256(raw).hexdigest()
    args.output.parent.mkdir(parents=True, exist_ok=True)
    args.output.write_text(json.dumps(report, indent=2) + "\n", encoding="utf-8")
    print(report["disposition"], report["counts"])
    print("Review record:", args.output)
    return 2 if report["disposition"] == "HOLD_FOR_TRIAGE" else 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Commands 20.4 — Create the triage summary and run parser checks

```
python review/review_image.py evidence/scan.json evidence/review.json
python -m unittest tests.test_review_image -v
```

A nonzero exit status of 2 here means HOLD_FOR_TRIAGE. This is a deliberate outcome, not a broken Python tool. Review evidence/review.json and the underlying report rather than rerunning until the command happens to pass.

File: tests/test_review_image.py — synthetic tests; not evidence of a clean real image

```
"""Synthetic fixtures test the report parser, not image safety."""
import unittest
from review.review_image import summarize


class ReviewImageTests(unittest.TestCase):
    def test_high_finding_creates_hold(self):
        report = {"ArtifactName": "deployboard:sample", "Results": [
            {"Vulnerabilities": [{"Severity": "HIGH", "VulnerabilityID": "CVE-TEST-1",
                "PkgName": "example", "InstalledVersion": "1", "FixedVersion": "2"}]}]}
        outcome = summarize(report)
        self.assertEqual(outcome["disposition"], "HOLD_FOR_TRIAGE")
        self.assertEqual(outcome["counts"]["HIGH"], 1)

    def test_empty_findings_still_require_human_review(self):
        result = summarize({"ArtifactName": "deployboard:sample", "Results": []})
        self.assertEqual(result["disposition"], "MANUAL_REVIEW_REQUIRED")

    def test_missing_report_fields_cannot_pass(self):
        with self.assertRaises(ValueError):
            summarize({"Results": []})


if __name__ == "__main__":
    unittest.main()
```

Observed condition	Recommended disposition
Fixable CRITICAL or HIGH	Hold; patch or replace the component and rescan.
Unfixed but potentially exploitable	Hold pending mitigation and a documented decision.
Likely not affected	Request evidence of non-impact and a reviewer sign-off.
No reported HIGH/CRITICAL	Continue human review; other release gates still apply.

20.5 Generate and Inspect a Software Bill of Materials

A Software Bill of Materials (SBOM) inventories components found in an artifact. It helps you answer “Which dependencies did we ship?” and provides a useful reference when a new advisory appears after release. An SBOM is not a vulnerability scan and its completeness depends on the tool, language ecosystems, and artifact visibility.

Commands 20.5 — Generate a CycloneDX SBOM and verify valid JSON

```
trivy image --format cyclonedx --output evidence/sbom.cdx.json deployboard:ch20-review
python -m json.tool evidence/sbom.cdx.json
```

In the CycloneDX JSON look for bomFormat, specVersion, metadata, and components. Packages listed for the final image can differ from packages used temporarily by a build stage. Compare the inventory against requirements.txt and note packages that are direct or transitive dependencies.

A controlled source inspection

Commands 20.6 — Connect package inventory to your local image

```
docker image history --no-trunc deployboard:ch20-review
docker image inspect deployboard:ch20-review --format "{{.Id}}"
```

The image ID reported locally is useful for the local evidence sheet; it does not replace a registry digest once an image is published. Keep the scan report and SBOM in the protected evidence folder, especially if they reveal sensitive internal package names or software versions.

20.6 Understand Build Provenance and Attestations

Provenance records how a build was produced: builder identity, inputs, and other build properties. An attestation is machine-readable evidence attached to a build artifact; it is not automatically a trustworthy claim simply because it exists. Docker Buildx can attach SBOM and provenance attestations to an image index when pushing to a registry.

For an optional registry exercise, use a namespace you control. Do not type the placeholder verbatim and do not publish private project code without permission. First authenticate with docker login in a way appropriate for your environment.

Command 20.7 — Optional registry build with SBOM and minimal provenance attestations

```
docker buildx build --provenance=mode=min --sbom=true \
  --metadata-file evidence/build-metadata.json \
  --tag YOUR_NAMESPACE/deployboard:20.0.0 --push .
```

The preceding multiline command uses Bash line continuation; run it in Bash or WSL. In PowerShell, place its options on one line. If you use a classic Docker image store, locally loading an image can discard attestations. Pushing with Buildx retains them in registries supporting OCI image indices. Docker notes that max-level provenance can expose build arguments and other build detail. Never put credentials into ARG or ENV; keep using Chapter 19 secret mounts.

Commands 20.8 — Optional: inspect the pushed registry artifact and its provenance

```
docker buildx imagetools inspect YOUR_NAMESPACE/deployboard:20.0.0
docker buildx imagetools inspect YOUR_NAMESPACE/deployboard:20.0.0 \
  --format "{{json .Provenance.SLSA}}"
```

Output formats depend on build and builder versions; inspect the manifest first and verify the attestation you expect is actually present. If the provenance output is empty, record that as missing evidence and investigate rather than asserting it was generated.

20.7 Use Digests to Link Deployed Images to Releases

A tag can be repointed; a digest identifies registry content. Save the top-level image-index digest when recording a multi-platform release and distinguish it from the per-platform manifest digests. A release record must

bind one source revision, one reviewed artifact digest, the associated scan, and the approval decision.

Commands 20.9 — Optional: record the immutable registry digest and index metadata

```
docker buildx imagetools inspect YOUR_NAMESPACE/deployboard:20.0.0
docker buildx imagetools inspect YOUR_NAMESPACE/deployboard:20.0.0 --raw
```

For a deployment use YOUR_NAMESPACE/deployboard@sha256:<actual_digest> after substituting the verified registry digest. This chapter does not overwrite the existing local Compose stack or delete named volumes. Chapter 25 covers registries and Chapter 28 covers applying pinned release references to production Compose configurations.

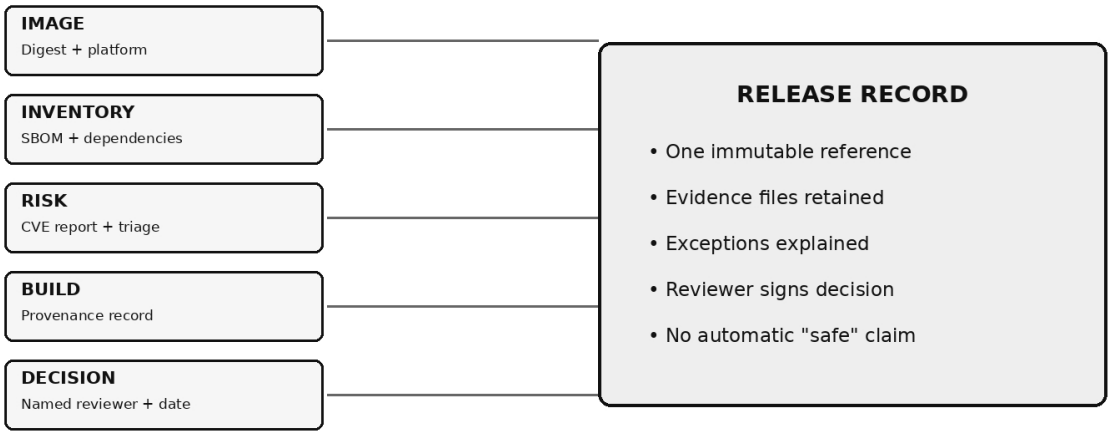


Figure 20.2 — A release record binds one immutable artifact to inventory, risk assessment, provenance, and a named decision.

20.8 Explain Image Signing and Verification Options

Signing allows a verifier to check that an identified artifact has an associated cryptographic signature made by an expected signer. It does not prove there are no vulnerabilities, that the builder was uncompromised, or that an attached SBOM is complete. Cosign supports key-based and identity-based signing; the right verification policy depends on how your organization establishes and protects signer identity.

Choose a verification policy before choosing a signing command

- Verify the exact digest rather than accepting a mutable tag as the subject.

- Use a documented trusted signer identity or key and expected issuer; do not accept any signer.
- Require signature verification to fail closed in the deployment gate.
- Keep signing credentials away from untrusted builds and temporary CI environments.

This chapter deliberately explains the options without generating real signing keys or pretending that a demonstration signature establishes organizational trust. Use the official Sigstore Cosign documentation to select a verified workflow for your registry and identity provider.

20.9 Prevent Untrusted Code from Reaching Privileged Pipelines

The quickest way to undermine image provenance is to let unreviewed contributor code execute in a privileged publishing job. A pull request can modify a Dockerfile, build script, dependency definition, or test fixture. Treat those files as executable input and separate untrusted verification from any process that can push images or access production secrets.

Untrusted verification job	Privileged publishing job
Read-only token and no deployment secrets	Least-privilege registry publishing credential.
Runs proposed Dockerfile and tests in isolation	Uses approved source after review and checks.
Produces reports, never pushes official tags	Publishes only from approved branches or tags.
No reusable privileged host Docker socket	Ephemeral, isolated runner; restricted network access.

GitHub specifically warns against checking out and executing untrusted pull-request code in a pull_request_target workflow that has secret access. Chapter 26 implements CI safely. For now, the enforceable local rule is simple: a scan report from an unreviewed build cannot authorize publication.

20.10 Document a Repeatable Image Review Procedure

A good release procedure is boring to repeat and difficult to falsify. Use this sequence every time the base image, application files, Python dependencies,

or build configuration changes. Store evidence securely and never edit reports to make the outcome look better.

Repeatable review procedure

- Identify the exact source revision or local source-file hash and inspect changes since the prior release.
- Check the base-image origin, selected architecture, and resolved digest.
- Build a candidate image and capture the immutable local image ID.
- Produce Trivy JSON and a CycloneDX SBOM from the candidate.
- Triage HIGH and CRITICAL findings; retain decisions and remediation notes.
- Optionally publish an approved candidate with provenance and SBOM attestations.
- Record the registry digest, reviewer, date, and explicit approval or hold.

The supplied review/release-record.template.json is intentionally NOT_APPROVED. Do not replace placeholders with invented digests, claim a signature was checked when it was not, or turn a missing scan into an exception.

File: review/release-record.template.json — complete release decision template

```
{
 "release": "20.0.0-candidate",
 "image_reference": "REGISTRY/REPOSITORY@sha256:REPLACE_WITH_ACTUAL_DIGEST",
 "source_revision": "REPLACE_WITH_VERIFIED_REVISION_OR_LOCAL_SOURCE_HASH",
 "source_reviewed": false,
 "base_image_digest": "REPLACE_WITH_INSPECTED_DIGEST",
 "scan_summary_file": "evidence/review.json",
 "sbom_file": "evidence/sbom.cdx.json",
 "provenance_checked": false,
 "signature_verified": false,
 "exceptions": [],
 "reviewer": "",
 "review_date": "",
 "decision": "NOT_APPROVED"
}
```

For long-term operation, establish an update cadence and a vulnerability recheck process: a release that was reviewed last month may need a new assessment when advisories change. Keep previous records rather than overwriting them.

20.11 Checkpoint: Approve a Traceable Image Release

Your challenge is to produce an evidence packet that another engineer can verify without relying on your memory. You may complete the local path with NO registry account; if you also push to a registry, attach the digest and attestation inspection results.

Run the local gate

Commands 20.10 — Rebuild, inventory, triage, and test from a clean evidence directory

```
docker buildx build --load -t deployboard:ch20-review .
trivy image --scanners vuln --format json --output evidence/scan.json deployboard:ch20-review
trivy image --format cyclonedx --output evidence/sbom.cdx.json deployboard:ch20-review
python review/review_image.py evidence/scan.json evidence/review.json
python -m unittest tests.test_review_image -v
```

Expected proof: the scan and SBOM files exist, the review summary identifies the actual scanned artifact, tests pass, and any HIGH/CRITICAL finding produces HOLD_FOR_TRIAGE. The exact vulnerability count is unknown until you run the scanner. Do not treat the absence of findings as blanket approval.

Release evidence	Pass condition
Source + base	Reviewed and identified by revision/hash and selected digest.
Image identity	Local ID recorded; registry digest if published.
Scanner + SBOM	Real JSON files generated and retained; valid format.
High-priority findings	Fixed, mitigated, or explicitly held with rationale.
Provenance + signature	Verified if claimed; otherwise marked not completed.
Final decision	Human-approved with evidence or explicitly NOT_APPROVED.

When something fails

A missing Trivy binary means installation is incomplete; a scanner database update failure means the scan has not completed; a builder without registry access cannot publish attestations; an empty .Provenance response means the expected attestation is not yet verified. Fix that stage and rerun it. Do not skip the evidence or erase the failure.

Before leaving the chapter, confirm that no new host ports were published, no secrets were copied into the image, and no existing database volumes were pruned. You have not “proved safety.” You have made an honest,

reproducible release decision possible. Chapter 21 turns this evidence discipline into application-level test gates.

PART VI

TEST, OBSERVE, AND IMPROVE RELIABILITY

Chapter 21 — Test Docker Applications Before You Ship

DeployBoard now runs as an authenticated API with PostgreSQL records, a Redis-backed job pipeline, a reverse proxy, and hardened containers. A successful local launch is encouraging; it is not release evidence. In this chapter you will create a separate testing environment that can fail safely, retain diagnostic information, and decide whether a release is ready to proceed.

You will leave the normal DeployBoard project untouched. Chapter 21 uses its own Compose project name, new database volume, generated testing credentials, and no host-published ports. The gate tests the application contract, not the security of the whole operating system. It is a local pre-release control, not a substitute for the later deployment checks.

What you will produce

A repeatable Python quality-gate command; a disposable test image; a real PostgreSQL/Redis integration test; a persistence check across API recreation; a nonzero-exit test; and a release evidence worksheet. All new source files are printed in this chapter and included with the complete existing DeployBoard project in the accompanying file package.

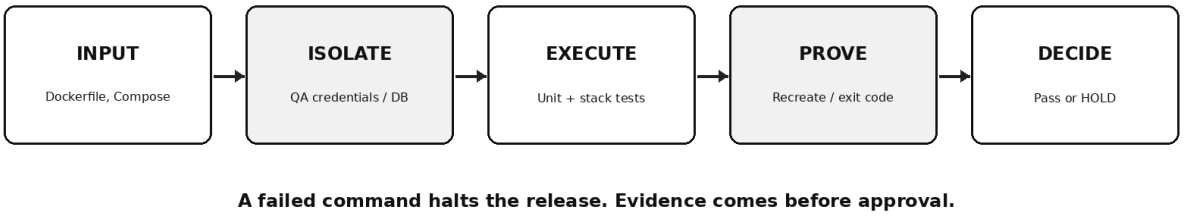


Figure 21.1 — Release testing is a sequence of gates, not a green status inferred from container startup.

21.1 Distinguish Unit, Integration, Smoke, and Deployment Tests

Test categories answer different questions. A unit test can validate status-transition logic without launching Docker; an API integration test exercises authentication, storage, and HTTP responses; a smoke test checks a running service through a small number of high-value requests; a deployment test verifies that the configured services cooperate in their intended environment.

Evidence type	What it proves	DeployBoard example
Unit / contract	A bounded function or API rule behaves as specified	Bad transitions return HTTP 409
Containerized tests	The test environment installs and runs declared dependencies	pytest exits zero inside a disposable image
Integration	Multiple real components communicate	API → PostgreSQL and Redis worker
Smoke / deployment	A started deployment handles basic operations	/ready returns 200; protected write succeeds
Recovery	Data and application recover from planned disruption	Recreated API still sees stored record

A single green check is never enough: /health can return HTTP 200 while the database is unavailable. The /ready endpoint is stronger evidence of the configured database, but it is not a comprehensive guarantee of future availability or worker success.

21.2 Run Tests Inside Consistent Container Environments

The test container must have its own development dependencies, yet it must not be shipped as the production runtime image. Use a dedicated Dockerfile so pytest and httpx are installed only in the disposable test image. The Dockerfile-specific ignore rules include tests while excluding credentials and local artifacts.

File: Dockerfile.test

```
# A disposable image only for tests. Not a production runtime image.
FROM python:3.13-slim
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
WORKDIR /service
COPY requirements.txt requirements-dev.txt ./
RUN pip install --no-cache-dir -r requirements-dev.txt
COPY app ./app
COPY migrations ./migrations
COPY alembic.ini ./alembic.ini
COPY tests ./tests
COPY review ./review
```

```
COPY tools ./tools
CMD ["python", "-m", "pytest", "-q", "-m", "not stack", "tests"]
```

File: Dockerfile.test.dockerignore

```
# Dockerfile-specific ignore rules: tests must be included.
.git
.env
.env.*
!.env.example
private/
qa-private/
__pycache__/
*.pyc
.pytest_cache/
*.db
*.zip
site/
proxy/
evidence/
```

The existing application image continues to use Dockerfile and .dockerignore. Changing the production image just to add pytest would increase its size and unnecessary attack surface. Confirm test isolation before building: the Dockerfile.test image is intended for QA, not distribution.

21.3 Create a Disposable Test Database and Sample Records

Compose names volumes using its project identity. Running tests with the original project name would risk operating on genuine learning data. Instead, the QA overlay changes the image tags, excludes the optional proxy, defines a test service, and points secrets to newly generated files. The Chapter 18 hardening override is also included, so the application services remain subject to their agreed runtime restrictions. The base schema still comes from the same Alembic migrations.

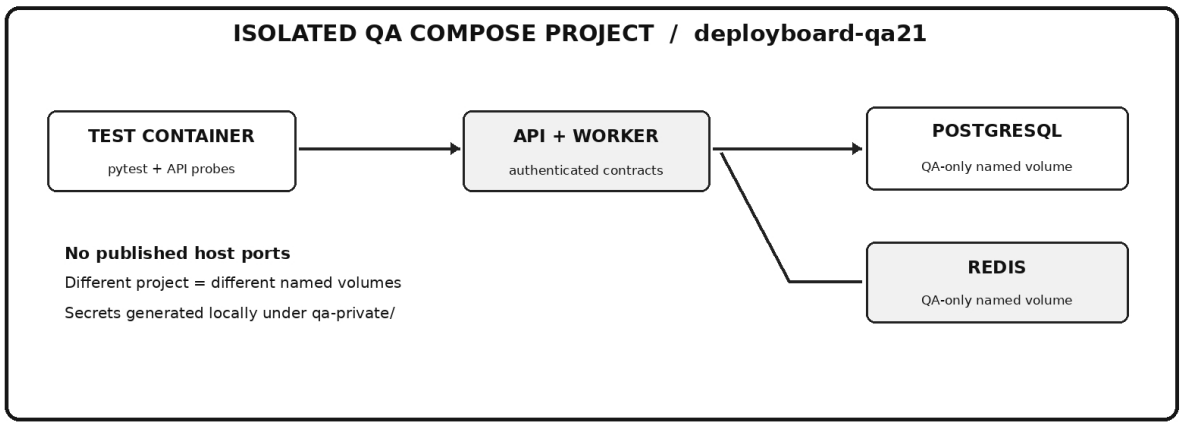


Figure 21.2 — The quality-gate network and volumes are separate from the normal DeployBoard stack.

File: compose.qa.yaml (additive overlay)

```
# Overlay: use base + hardening + QA files, with -p deployboard-qa21.
# Dedicated project name isolates volumes from the chapter 14-20 stack.
services:
 api:
  image: deployboard:ch21-qa
 publisher:
  image: deployboard:ch21-qa
 worker:
  image: deployboard:ch21-qa
 migrate:
  image: deployboard:ch21-qa
 proxy:
  # Do not start the proxy during this local quality gate.
  # Its base published port could conflict with the existing stack.
  profiles: [qa-preview]
 tests:
  build:
   context: .
   dockerfile: Dockerfile.test
  image: deployboard:ch21-tests
  environment:
   DB_HOST: db
   DB_USER: deployboard
   DB_NAME: deployboard
   DB_PASSWORD_FILE: /run/secrets/db_password
   REDIS_HOST: redis
   DEPLOYBOARD_VIEWER_TOKEN_FILE: /run/secrets/viewer_token
   DEPLOYBOARD_OPERATOR_TOKEN_FILE: /run/secrets/operator_token
   API_URL: http://api:8000
  secrets: [db_password, viewer_token, operator_token]
  depends_on:
   api: {condition: service_healthy}
  networks: [private]
  profiles: [qa-tools]
  restart: 'no'
secrets:
 db_password:
  file: ./qa-private/db_password.txt
 viewer_token:
  file: ./qa-private/viewer_token.txt
```

```
operator_token:
  file: ./qa-private/operator_token.txt
```

Generate dedicated QA-only credentials with the following helper. It never overwrites an existing secret. The generated files are untracked, and their values are never included in the manuscript or the ZIP.

File: tools/prepare_qa_secrets.py

```
"""Create separate lab credentials. Never overwrite existing files."""
from pathlib import Path
import os
import secrets

ROOT = Path(__file__).resolve().parents[1]
DIRECTORY = ROOT / 'qa-private'
FILES = ('db_password.txt', 'viewer_token.txt', 'operator_token.txt')


def prepare() -> None:
    if DIRECTORY.is_symlink():
        raise RuntimeError('Refusing a symlinked QA secrets directory')
    DIRECTORY.mkdir(mode=0o700, exist_ok=True)
    if os.name == 'posix':
        DIRECTORY.chmod(0o700)
    for name in FILES:
        path = DIRECTORY / name
        if path.is_symlink():
            raise RuntimeError(f'Refusing symlinked secret: {name}')
        if not path.exists():
            path.write_text(secrets.token_urlsafe(32) + '\n', encoding='utf-8')
        if os.name == 'posix':
            path.chmod(0o444)
    print('QA credential files ready; values not displayed')


if __name__ == '__main__':
    prepare()
```

From the Chapter 21 source directory, run the two commands below. The cross-platform helper always applies all three Compose files with the dedicated QA project name. Windows PowerShell users may use py instead of python. If Docker Desktop is installed, ensure the Linux container engine is active.

File: tools/qa_compose.py (cross-platform command shorthand)

```
"""Run Compose with the dedicated QA project and both config files."""
from pathlib import Path
import subprocess
import sys

ROOT = Path(__file__).resolve().parents[1]
PREFIX = ['docker', 'compose', '-p', 'deployboard-qa21',
          '-f', 'compose.yaml', '-f', 'compose.harden.yaml',
          '-f', 'compose.qa.yaml']
if __name__ == '__main__':
    if len(sys.argv) == 1:
```

```
    raise SystemExit('Provide a Compose subcommand')
  raise SystemExit(subprocess.call(PREFIX + sys.argv[1:], cwd=ROOT))
```

Prepare and validate without starting containers

```
python tools/prepare_qa_secrets.py
python tools/qa_compose.py config -q
```

Expected result: “QA credential files ready; values not displayed,” followed by no error from Compose. The external deployment remains unaffected. The QA database is still persistent within its own named volume so a recreation test has something real to measure.

21.4 Write API Integration Tests for DeployBoard

Write integration tests against the running QA API, not a mocked SQLite repository. Use unique deployment names so two runs cannot collide, a viewer token for read-only access, and an operator token for mutations. Always delete the test deployment in a finally block, even if an assertion fails. The test includes a background-job request and bounded polling; it fails rather than waiting forever when a worker is unhealthy.

File: pytest.ini

```
[pytest]
markers =
  stack: requires the live isolated Compose quality-gate stack
```

File: tests/test_stack_live.py

```
"""Execute only inside isolated QA Compose: pytest -m stack."""
import os
import time
from uuid import uuid4
import httpx
import pytest
from app.settings import Settings

pytestmark = pytest.mark.stack
URL = os.getenv('API_URL', 'http://api:8000').rstrip('/')


def call(client, method, path, token=None, **kwargs):
  headers = {'Authorization': f'Bearer {token}'} if token else {}
  return client.request(method, URL + path, headers=headers, **kwargs)


def test_real_database_auth_crud_and_background_job():
  conf = Settings.from_env()
  assert conf.valid_tokens and conf.valid_database
  marker = f'QA-{uuid4().hex[:10]}'
  with httpx.Client(timeout=8.0) as client:
    assert call(client, 'GET', '/health').status_code == 200
```

```
    assert call(client, 'GET', '/ready').status_code == 200
    assert call(client, 'GET', '/v1/deployments').status_code == 401
    payload = {'name': marker, 'environment': 'staging', 'version': '21.0'}
    assert call(client, 'POST', '/v1/deployments',
        token=conf.viewer_token, json=payload).status_code == 403
    invalid = call(client, 'POST', '/v1/deployments',
        token=conf.operator_token, json={**payload, 'unexpected': True})
    assert invalid.status_code == 422
    created = call(client, 'POST', '/v1/deployments',
        token=conf.operator_token, json=payload)
    assert created.status_code == 201, created.text
    item_id = created.json()['id']
    try:
        path = f'/v1/deployments/{item_id}'
        assert call(client, 'GET', path, token=conf.viewer_token).status_code == 200
        assert call(client, 'PATCH', path, token=conf.operator_token,
            json={'version': '21.1'}).json()['version'] == '21.1'
        event = call(client, 'POST', path + '/events', token=conf.operator_token,
            json={'status': 'running', 'note': 'QA transition'})
        assert event.status_code == 201
        assert call(client, 'POST', path + '/events', token=conf.operator_token,
            json={'status': 'pending'}).status_code == 409
        job = call(client, 'POST', path + '/checks?key=quality-gate-2101',
            token=conf.operator_token)
        assert job.status_code == 202, job.text
        job_id = job.json()['id']
        latest = None
        for attempt in range(40):
            latest = call(client, 'GET', f'/v1/checks/{job_id}',
                token=conf.viewer_token)
            assert latest.status_code == 200
            if latest.json()['status'] in ('succeeded', 'failed'):
                break
            time.sleep(1)
        assert latest.json()['status'] == 'succeeded', latest.text
        duplicate = call(client, 'POST', path + '/checks?key=quality-gate-2101',
            token=conf.operator_token)
        assert duplicate.json()['id'] == job_id
    finally:
        result = call(client, 'DELETE', f'/v1/deployments/{item_id}',
            token=conf.operator_token)
        assert result.status_code == 204, result.text
```

The default Chapter 20 tests still verify API contracts and secret handling against disposable local fixtures. Their SQLite databases do not prove PostgreSQL connectivity; that is the responsibility of the explicitly marked stack test above. If the job completes as failed, the test preserves its HTTP response and prevents promotion.

21.5 Test Compose Service Health and Failure Recovery

Start only the QA infrastructure, migrate it, and then start the application tier. Do not run the default Compose project or rely on a previous database

volume. Compose dependency conditions can delay API startup until PostgreSQL is healthy; they do not provide an orchestrator that automatically repairs arbitrary dependency failures later.

Start the isolated test stack (run these commands individually)

```
python tools/qa_compose.py up -d --wait db redis
python tools/qa_compose.py run --rm migrate
python tools/qa_compose.py up -d --wait api publisher worker
```

Inspect health rather than simply seeing “running”: use docker compose ... ps and docker compose ... logs --tail 30 api worker. In a deliberately isolated lab, stop the QA database and observe that the running API becomes unready or reports database failures; restore the DB and verify readiness again. Do not run this fault experiment against the normal project.

Optional controlled fault; restore the database immediately

```
python tools/qa_compose.py stop db
python tools/qa_compose.py start db
python tools/qa_compose.py ps
```

Expected behavior is conditional: the exact temporary HTTP response depends on connection reuse, timeouts, and health-check intervals. What matters is that the gate detects loss of readiness and successful recovery—not an invented fixed number of restart seconds.

21.6 Check Volume Behavior with Safe Test Fixtures

The database lives in the QA project volume, not in the API container. Create a uniquely identifiable QA marker through the API, replace only the API container, and verify that the record remains. This tests storage independently of the Chapter 10 PostgreSQL exercise and does not alter the original named volume.

File: tools/persistence_probe.py

```
"""Seed or verify one marker through the real API in QA. Do not delete it."""
import argparse
import os
import urllib.request
import json
from app.settings import Settings

URL = os.getenv('API_URL', 'http://api:8000').rstrip('/')
LABEL = 'QA-CH21-PERSISTENCE-RECREATE'


def call(path, method='GET', data=None):
    cfg = Settings.from_env()
```

```
    body = json.dumps(data).encode() if data is not None else None
    req = urllib.request.Request(URL + path, data=body, method=method,
        headers={'Authorization': 'Bearer ' + cfg.operator_token,
                 'Content-Type': 'application/json'})
    with urllib.request.urlopen(req, timeout=10) as r:
        return json.load(r)


def main():
    p = argparse.ArgumentParser()
    p.add_argument('phase', choices=['seed', 'verify'])
    phase = p.parse_args().phase
    matching = [r for r in call('/v1/deployments') if r['name'] == LABEL]
    if phase == 'seed' and not matching:
        call('/v1/deployments', 'POST',
             {'name': LABEL, 'environment': 'development', 'version': '21.0'})
        print('Created QA persistence marker')
    if phase == 'verify':
        assert matching, 'Persistent record missing after API recreation'
        print('PASS: same QA PostgreSQL record survived API recreation')


if __name__ == '__main__':
    main()
```

Prove survival across API recreation

```
python tools/qa_compose.py run --rm tests python tools/persistence_probe.py seed
python tools/qa_compose.py up -d --no-deps --force-recreate api
python tools/qa_compose.py run --rm tests python tools/persistence_probe.py verify
```

Expected result: “PASS: same QA PostgreSQL record survived API recreation.” The test preserves the marker to support repeated verification; you can remove the QA volume only when you intentionally discard QA data. Never use docker compose down -v without verifying the targeted project name.

21.7 Validate Dockerfile and Compose Configuration

Run both static configuration checks before the expensive container tests. docker compose config -q validates and merges the YAML; docker build --check . checks the production Dockerfile and build inputs without producing a runtime image. Neither command replaces a functional build, nor does a clean result prove that security policies are sufficient.

Configuration and real image build checks

```
python tools/qa_compose.py config -q
docker build --check .
python tools/qa_compose.py build api tests
```

The tool versions shown by docker compose version and docker buildx version should be recorded in the evidence worksheet. Build checks and Compose schemas change over time, so cite the exact versions that produced your result.

21.8 Test Failure Paths and Nonzero Exit Codes

A release gate is only as reliable as its handling of failures. A broken command that prints ERROR but returns exit 0 can be mistakenly accepted by a naive script. The small probe below intentionally exits with code 23. The quality gate treats only that expected code as success for this negative test.

File: tools/exit_probe.py

```
"""A deliberately failing task for checking exit-status propagation."""
raise SystemExit(23)
```

Expected: container command exits with code 23, not zero

```
python tools/qa_compose.py run --rm tests python tools/exit_probe.py
```

Other important negative outcomes include HTTP 401 without a token, HTTP 403 for a viewer attempting to write, HTTP 422 for invalid input, and HTTP 409 for a forbidden status transition. The live test asserts each contract; it does not merely print a status line.

21.9 Avoid Test Pollution Between Runs

Repeatability means your tests do not depend on yesterday’s database contents or another author’s active containers. Use a dedicated project name, unique record markers, an isolated credential directory, and a generated QA database volume. Each live test cleans up its own deploy record, while the persistence marker remains intentionally until QA data is removed.

Resource	Isolation rule	Safe disposal
QA containers	Project: deployboard-qa21	compose down (without -v)
QA named volumes	Owned by that QA project	Retain until explicitly discarded
QA credentials	qa-private/ ignored by VCS and builds	Rotate or delete deliberately
Test records	Unique marker; delete in finally	Never delete shared application rows

Resource	Isolation rule	Safe disposal
Real DeployBoard	Leave its original Compose project untouched	No test cleanup commands target it

Do not put actual secrets, dumps, CI tokens, or private vulnerability findings into the published evidence archive. Record success/failure and tool versions instead.

21.10 Create a Repeatable Pre-Release Test Command

The one-command gate calls Docker with Python subprocess rather than a Bash-only shell pipeline. It halts on any unexpected nonzero exit code. By default it preserves QA containers on failure so the logs remain available; the optional --cleanup flag stops QA containers only after success and does not delete named volumes.

File: tools/quality_gate.py

```
"""Run the isolated DeployBoard Chapter 21 gate. Requires Docker Engine."""
import argparse
from pathlib import Path
import subprocess
import sys
from prepare_qa_secrets import prepare

ROOT = Path(__file__).resolve().parents[1]
C = ['docker', 'compose', '-p', 'deployboard-qa21',
    '-f', 'compose.yaml', '-f', 'compose.harden.yaml',
        '-f', 'compose.qa.yaml']


def run(args, expected=0):
    printable = ' '.join(args)
    print('\n$ ' + printable, flush=True)
    outcome = subprocess.run(args, cwd=ROOT, check=False)
    if outcome.returncode != expected:
        raise RuntimeError(f'Unexpected exit {outcome.returncode}; expected {expected}: ' + printable)


def gate(cleanup=False):
    prepare()
    run(['docker', 'version', '--format', '{{.Server.Version}}'])
    run(['docker', 'compose', 'version'])
    run(C + ['config', '-q'])
    run(['docker', 'build', '--check', '.'])
    run(C + ['build', 'api', 'tests'])
    run(C + ['up', '-d', '--wait', '--wait-timeout', '120', 'db', 'redis'])
    run(C + ['run', '--rm', 'migrate'])
    run(C + ['up', '-d', '--wait', '--wait-timeout', '120',
            'api', 'publisher', 'worker'])
    run(C + ['run', '--rm', 'tests', 'python', '-m', 'pytest', '-q',
            '-m', 'not stack', 'tests'])
    run(C + ['run', '--rm', 'tests', 'python', '-m', 'pytest', '-q',
```

```
            '-m', 'stack', 'tests/test_stack_live.py'])
    run(C + ['run', '--rm', 'tests', 'python', 'tools/persistence_probe.py', 'seed'])
    run(C + ['up', '-d', '--no-deps', '--force-recreate', 'api'])
    run(C + ['run', '--rm', 'tests', 'python', 'tools/persistence_probe.py', 'verify'])
    run(C + ['run', '--rm', 'tests', 'python', 'tools/exit_probe.py'], expected=23)
    print('\nPASS: local quality gate. Release still requires human review.')
    if cleanup:
        run(C + ['down'])  # Deliberately keeps named QA volumes.


if __name__ == '__main__':
    a = argparse.ArgumentParser()
    a.add_argument('--cleanup', action='store_true',
                   help='stop QA containers after success; preserve named volumes')
    x = a.parse_args()
    try:
        gate(x.cleanup)
    except (RuntimeError, OSError) as exc:
        print('HOLD:', exc, file=sys.stderr)
        print('QA resources preserved for debugging; never run down -v blindly.',
              file=sys.stderr)
        raise SystemExit(1)
```

Run the quality gate; optional safe cleanup after success

```
python tools/quality_gate.py
python tools/quality_gate.py --cleanup
```

The required sequence is configuration validation, Dockerfile checks, isolated image builds, database migration, hardened service startup, existing tests, a real stack integration test, API recreation with persistent storage verification, and an expected nonzero-exit check. A failure leaves the release on HOLD. A pass means these specified checks completed—not that Docker, the host, or all external integrations are guaranteed safe.

21.11 Checkpoint: Run the Complete Local Quality Gate

Run the gate from a clean Chapter 21 project directory. Verify the command exits zero, then capture your acceptance evidence. Do not mark steps as passed unless you have observed the corresponding result in your own environment.

Acceptance proof	Pass standard
Isolated project	QA containers and volumes use deployboard-qa21; no host ports
Static checks	Compose configuration and Docker build checks exit 0
Automated contract tests	Unit/API/secret/review tests finish without failures

Acceptance proof	Pass standard
Live integration	Authenticated API, PostgreSQL, and Redis job succeed
Persistence	Marker survives api --force-recreate
Failure propagation	Expected exit 23 is detected; unexpected codes fail gate
Resource safety	Normal DeployBoard project/volumes unchanged
Record	Version, date, command, status, and unresolved issues documented

If the gate fails

Use the last printed command and relevant logs; retain the QA volumes, repair only the failing condition, and rerun from the first check. Never “fix” a failing release by deleting the real database or suppressing an assertion.

What comes next

Chapter 22 adds deeper signals: container events, logs, resource measurements, application correlations, and useful health diagnostics. The quality gate from this chapter remains your baseline when later reliability changes are tested.

Chapter 22 — Observe Containers, Services, and Resource Usage

A healthy API response is not a guarantee that every part of a distributed application is doing useful work. You will follow one deployment request through Caddy, FastAPI, PostgreSQL, Redis, and the worker; collect evidence; then isolate and repair a controlled background-processing failure.

YOU WILL BUILD	PROOF YOU WILL COLLECT
Structured request and worker logs	Traceable request IDs and job IDs, without printing credentials
An additive Compose observability layer	Rotating local container logs, service health and resource snapshots
A reproducible incident drill	A queued check survives Redis downtime and completes after recovery

Before you begin

Use the DeployBoard project completed in Chapter 21 on a local Docker Engine with Docker Compose v2, Python 3, and available disk space. Work from the project root, with the Chapter 19 local credentials already prepared. The source ZIP accompanying this chapter contains the prior project and every new file. The book listings show all new code; unchanged project files remain as taught in Chapters 13–21.

Safety boundary: the Compose project remains deployboard-ch14, retaining its existing named volumes. The proxy publishes only 127.0.0.1:8080. Do not run docker compose down -v, remove Docker volumes, expose Redis/PostgreSQL ports, or reproduce incidents against important data. Treat logs as sensitive operational records.

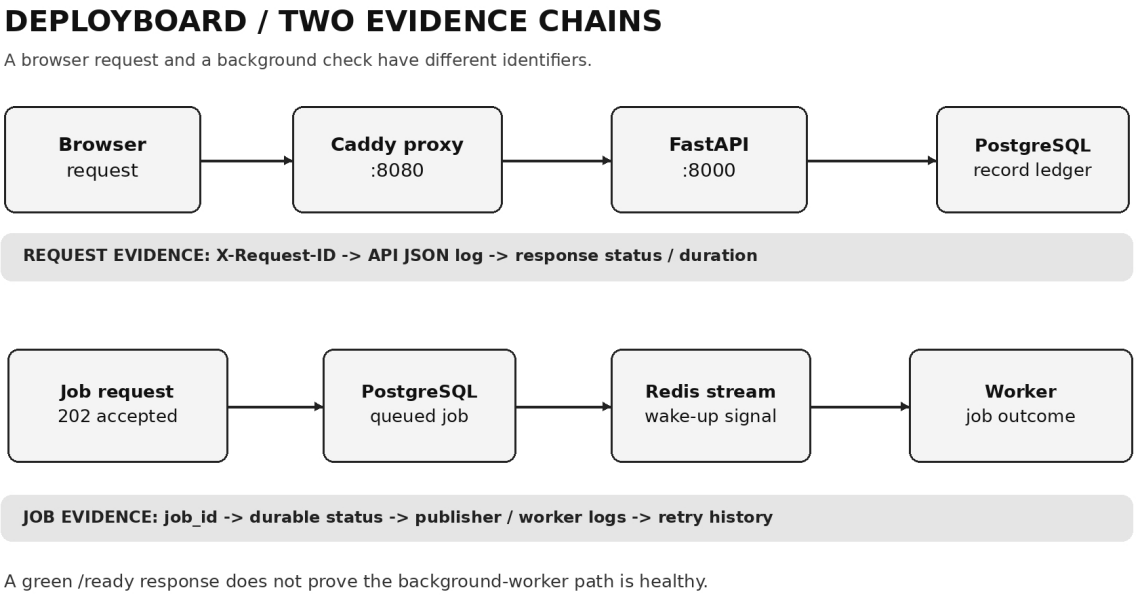


Figure 22.1 Two separate but complementary evidence chains: HTTP request ID and durable job ID.

22.1 Read Structured Application Logs

Logs should help you reconstruct events, not merely announce that a process is alive. Each DeployBoard application event needs a timestamp (UTC), severity, component, stable event name, and selected context. Avoid dumping requests, Authorization headers, secret-file values, SQL connection URLs, or complete exception objects. The application and worker share the same JSON formatter while PostgreSQL and Redis keep their native log formats.

Create app/observability.py

This module emits compact one-line JSON events to stdout. The formatter permits only named extra attributes. An incoming X-Request-ID is accepted only when it matches a restricted pattern; otherwise a new identifier is generated. That validates the *identifier*, not its trustworthiness as an identity credential.

APP/OBSERVABILITY.PY

```
"""Small, secret-conscious JSON observability layer for DeployBoard."""
import json
import logging
import re
import sys
from contextvars import ContextVar
from datetime import datetime, timezone
```

```
from uuid import uuid4

_current_request_id = ContextVar("deployboard_request_id", default="-")
_ID = re.compile(r"^[A-Za-z0-9._-]{8,64}$")
_ALLOWED = ("event", "status", "duration_ms", "job_id", "attempts")


def request_id_from_header(candidate: str | None) -> str:
    """Do not reflect arbitrary header text into logs or response headers."""
    if candidate and _ID.fullmatch(candidate):
        return candidate
    return uuid4().hex


class JSONLogFormatter(logging.Formatter):
    """Only explicitly selected fields; never serialize request or env objects."""
    def format(self, record):
        entry = {
            "ts": datetime.fromtimestamp(record.created, timezone.utc)
                  .isoformat(timespec="milliseconds").replace("+00:00", "Z"),
            "level": record.levelname,
            "component": record.name,
            "message": record.getMessage(),
        }
        rid = _current_request_id.get()
        if rid != "-":
            entry["request_id"] = rid
        for key in _ALLOWED:
            if hasattr(record, key):
                entry[key] = getattr(record, key)
        return json.dumps(entry, separators=(",", ":"), ensure_ascii=True)


def configure_logging():
    """Do not install multiple handlers during tests or app construction."""
    root = logging.getLogger("deployboard")
    root.setLevel(logging.INFO)
    root.propagate = False
    if not any(getattr(h, "_deployboard_json", False) for h in root.handlers):
        handler = logging.StreamHandler(sys.stdout)
        handler._deployboard_json = True
        handler.setFormatter(JSONLogFormatter())
        root.addHandler(handler)
    return root


def logger(component):
    configure_logging()
```

```
    return logging.getLogger("deployboard." + component)


def enter_request(rid):
    return _current_request_id.set(rid)


def leave_request(token):
    _current_request_id.reset(token)
```

In app/main.py, import time and request_id_from_header, logger, enter_request, leave_request. Immediately after api = FastAPI(...) inside create_app, add the following middleware. Keep every existing endpoint and permission dependency unchanged. The ZIP includes the entire updated app/main.py.

APP/MAIN.PY — INSERTION INSIDE CREATE_APP()

```
    audit = logger("api")
    @api.middleware("http")
    async def trace_requests(request: Request, call_next):
        rid = request_id_from_header(request.headers.get("X-Request-ID"))
        marker = enter_request(rid)
        started = time.perf_counter()
        try:
            response = await call_next(request)
            response.headers["X-Request-ID"] = rid
            audit.info("request_completed", extra={
                "event": request.method + " " + (request.scope["route"].path if request.scope.get("route")
else ("/v1/*" if request.url.path.startswith("/v1/") else "/other")),
                "status": response.status_code,
                "duration_ms": round((time.perf_counter() - started) * 1000, 1),
            })
            return response
        except Exception:
            # Do not place exception contents, URL queries, bodies, or headers in logs.
            audit.error("request_unhandled", extra={"event": "http_unhandled", "status": 500})
            raise
        finally:
            leave_request(marker)
```

After the change, /health still answers whether the API process can serve an HTTP request; /ready still checks its database-backed readiness contract. The middleware intentionally records a route template when available, never a URL query string. A request ID appears in the response header and

in the API JSON log. The ID is a correlation aid, not an access-control token.

Make worker and publisher logs parseable

```
# In app/worker.py and app/publisher.py:
from .observability import logger
LOG = logger("worker")      # app/worker.py
LOG = logger("publisher")   # app/publisher.py
# Remove their former logging.basicConfig / logging.getLogger setup.
# Avoid logging str(exc) for Redis errors; log the exception class instead.
```

DeployBoard workers already log a durable job ID when work succeeds; the publisher logs publication counts. Do not confuse the HTTP request ID with the job ID: one request may create a job that is processed later in a different container.

22.2 Use docker logs and docker compose logs Effectively

docker logs targets one container, whereas docker compose logs follows services belonging to this Compose project. Start with bounded time windows and line counts. Add --follow only while watching a live incident; press Ctrl+C to stop following without stopping containers.

TERMINAL — TARGETED LOG INSPECTION

```
python tools/dc22.py logs --since 10m --tail 60 api worker
python tools/dc22.py logs -f --tail 30 publisher worker
python tools/dc22.py ps
# From any terminal, obtain the current API container ID:
python tools/dc22.py ps -q api
# Then use: docker logs --since 5m --tail 40 <container-id>
```

Expected: API entries are one JSON object per line containing a request_completed event, status, and duration_ms. Worker and publisher entries are JSON; PostgreSQL and Redis output remain native text. Docker adds service prefixes and optional timestamps when you ask for them. If you see nothing, confirm the container is running and that its logging driver supports local log retrieval.

22.3 Inspect Container and Engine Events

Logs explain what an application says; engine events reveal what Docker did. Events are useful for correlating restarts, kills, health-status transitions, and container exits. They are not an indefinitely retained audit trail: capture them during an incident rather than assuming historical events will always be available.

TERMINAL — CONTAINER EVENTS AND STATE

```
docker events --since 15m --filter type=container
# In a second terminal, reproduce a safe service stop/start.
python tools/dc22.py ps --all
python tools/dc22.py ps -q api
# Substitute the returned ID; inspect only what you need:
docker inspect --format '{{json .State}}' <container-id>
```

Read State.Status, ExitCode, OOMKilled, and Health.Status when present. A running process may still be unready. Conversely, a container that is not configured with a health check has no useful health verdict. Never print the entire container inspect JSON in shared support messages: it can contain environment metadata.

22.4 Read docker stats and Resource Signals

docker stats is a short-term resource signal, not a historical profiler. Watch CPU%, memory usage versus its limit, network I/O, block I/O, and PIDs. CPU may exceed 100% on multicore hosts; memory percentage depends on cgroup accounting and a configured limit. A single busy sample alone should not trigger a restart.

TERMINAL — IMMEDIATE RESOURCE AND SERVICE SAMPLE

```
docker stats --no-stream
# Watch a short interval; stop the stream with Ctrl+C.
docker stats
python tools/observe_stack.py --cpu-warn 85 --mem-warn 85
```

SIGNAL	FIRST QUESTION	FOLLOW-UP
CPU elevated	Sustained work or looping process?	Compare worker job activity; collect several samples
Memory near limit	Is there a leak or known workload spike?	Check Docker events and OOMKilled flag
Block I/O high	Is PostgreSQL committing or swapping?	Check database logs and host disk space
Restart count rising	Is the main process crashing?	Inspect recent exit codes and startup logs

22.5 Design Health Checks for Actual Service Behavior

A good health check tests the capability the service is expected to provide at its own boundary. Keep the existing inexpensive FastAPI /health check, while /ready validates that tokens and PostgreSQL are configured and reachable. PostgreSQL uses pg_isready; Redis uses redis-cli ping. Neither check promises that queries, stream consumers, or the browser path will always work.

COMPOSE EXCERPT — API READINESS PROBE

```
# Existing compose.yaml behavior (excerpt; keep these checks):
api:
  healthcheck:
    test: ["CMD", "python", "-c",
      "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8000/ready', timeout=3)"]
    interval: 10s
    timeout: 4s
    retries: 3
    start_period: 20s
```

The example above is from the current project and is safe to retain; there is no need to duplicate it in compose.observe.yaml. Health checks execute inside the container, so they must not rely on a host-only address. Make each check bounded in time and free of credential output. “Unhealthy” is an observation, not an instruction to erase volumes or kill dependencies.

22.6 Separate Liveness, Readiness, and Dependency Availability

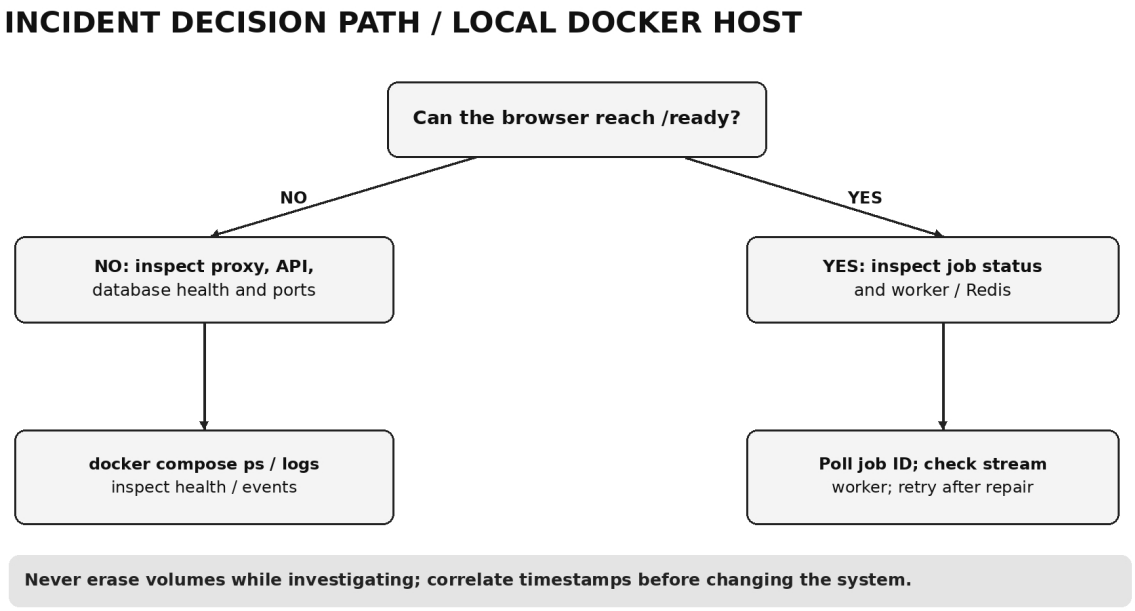


Figure 22.2 Triage by evidence: separate an unavailable API from a stalled background pipeline.

CHECK	WHAT IT PROVES	WHAT IT DOES NOT PROVE
GET /health	API process is responding	PostgreSQL, Redis, or workers work
GET /ready	API can use its PostgreSQL-backed store	Redis queue and worker make progress
docker compose ps	Container process and configured health status	Business task actually completed
Job status via /v1/checks/{id}	Persisted task lifecycle state	Why task is stuck without logs

A Compose depends_on: condition: service_healthy gate controls startup order. It does not continuously restart dependents when their dependencies become unhealthy, nor does it create full orchestrator-style failover. Redis availability is deliberately not a condition for read-only DeployBoard API readiness. A queued job can therefore remain unfinished while /ready returns 200.

22.7 Introduce Log Rotation and Retention

The Docker Engine json-file driver is often the default and may grow unbounded without explicit rotation. This workshop uses Docker’s local logging driver with per-container rotation and compression. Rotated history

is finite; it is not a backup and does not replace exporting logs when diagnosing significant incidents.

NEW COMPOSE.OBSERVE.YAML — COMPLETE FILE

```
# Chapter 22: add to compose.yaml and compose.harden.yaml (do not use alone).
# Per-service rotation applies to newly created containers only.
x-observe-logs: &observe_logs
  driver: local
  options:
    max-size: "10m"
    max-file: "3"

services:
  db:
    logging: *observe_logs
  redis:
    logging: *observe_logs
  api:
    logging: *observe_logs
  publisher:
    logging: *observe_logs
  worker:
    logging: *observe_logs
  proxy:
    logging: *observe_logs
  migrate:
    logging: *observe_logs
```

Apply this third Compose file after the baseline and hardened layers. A changed logging driver takes effect when the container is *recreated*, not retroactively for an existing container. Check the effective configuration before updating running services. The local driver’s max-size and max-file settings are intentionally bounded (10 MB × 3 files per service).

NEW TOOLS/DC22.PY — CROSS-SHELL COMPOSE LAUNCHER

```
"""Portable launcher for the Chapter 22 Compose file set."""
from pathlib import Path
import subprocess
import sys

ROOT = Path(__file__).resolve().parents[1]
COMMAND = ["docker", "compose", "-f", "compose.yaml",
           "-f", "compose.harden.yaml", "-f", "compose.observe.yaml"]
if __name__ == "__main__":
    raise SystemExit(subprocess.call(COMMAND + sys.argv[1:], cwd=ROOT))
```

TERMINAL — VALIDATE THEN APPLY

```
python tools/dc22.py config -q
python tools/dc22.py ps
python tools/dc22.py up -d --force-recreate api publisher worker proxy
# Never use "down -v" to apply logging changes.
```

If Compose shows a configuration merge error, stop before recreating anything. Never alter the daemon-wide logging default just to complete this chapter. Existing logs may be lost at recreation, so capture relevant evidence first.

22.8 Add Request Correlation and Useful Error Context

Exercise the correlation path without exposing any API token in a URL. Use a browser only for /health and /ready. For authenticated calls, use the Python incident probe provided later in the chapter, which reads a local secret file and never prints its contents.

TERMINAL — CORRELATE A REQUEST

```
python tools/incident_probe.py prepare
python tools/dc22.py logs --since 2m --tail 25 api
# Match request_id printed by prepare to API JSON log.
# Later: match job_id printed by submit with worker logs.
```

Expected: a PREPARED message with a deployment ID and request ID, followed by an API request_completed log event with that same ID and HTTP 201. Reject untrusted header values rather than echoing arbitrary text into log lines or HTTP responses. Log stable failure categories, safe identifiers, and HTTP status—not raw passwords, exception objects, or request bodies. Keep the request ID in incident notes as evidence, not as proof of user identity.

22.9 Observe API, Worker, Database, and Proxy Together

Use JSON Caddy access logs for consistent parsing: change format console to format json inside the existing proxy/Caddyfile log block. The route rules and bound address do not change.

UPDATED PROXY/CADDYFILE — COMPLETE FILE

```
:8080 {
  log {
```

```
    output stdout
    format json
  }

  @api path /health /ready /v1/* /docs /openapi.json
  handle @api {
    reverse_proxy api:8000
  }

  handle {
    root * /srv
    file_server
  }
}
```

View the stack as a chain of responsibilities: Caddy handles the local HTTP entry point; FastAPI processes authenticated commands; PostgreSQL persists deployments and durable job states; the publisher advertises committed jobs to Redis; the worker acknowledges them only after final state is recorded. Each tier has a different useful signal.

SERVICE	PRIMARY OBSERVATION	USEFUL NEXT CHECK
proxy	HTTP access/route status (JSON Caddy log)	Browser address and upstream API readiness
api	JSON request_completed and X-Request-ID	HTTP status and PostgreSQL-backed /ready
db	PostgreSQL startup, errors, persistence	Queued-job counts and disk health
redis	Stream transport, connectivity	Publisher/worker reconnect messages
publisher	Published committed-job count	Queued unpublished work in PostgreSQL
worker	job ID and final task status	Retry attempts and consumer progress

TERMINAL — THREE COMPLEMENTARY VIEWPOINTS

```
python tools/dc22.py logs --since 5m --tail 45 proxy api db
python tools/dc22.py logs --since 5m --tail 45 redis publisher worker
python tools/dc22.py run --rm migrate python tools/queue_snapshot.py
```

The queue snapshot reads counts from PostgreSQL and prints only status buckets; it does not export credentials or individual records. Use this to distinguish “API accepted a check” from “a worker completed that check.” For an accepted job, follow its job_id from the API response to the persisted status and finally to worker logs.

NEW TOOLS/QUEUE_SNAPSHOT.PY — FULL SOURCE

```
"""Print status counts from durable PostgreSQL queue, without credentials."""
from sqlalchemy import func, select
from app.db import background_jobs
from app.main import make_engine
from app.settings import Settings


def main():
    engine = make_engine(Settings.from_env())
    with engine.connect() as db:
        rows = db.execute(select(background_jobs.c.status, func.count())
            .group_by(background_jobs.c.status)).all()
        for status, count in sorted(rows):
            print(f"{status}: {count}")
    engine.dispose()


if __name__ == "__main__":
    main()
```

22.10 Set a Minimal Single-Host Alerting Approach

On one developer host, a small polling command is a reasonable first alarm, provided it has explicit thresholds and a nonzero exit status for unhealthy services. It is not a paging service, long-term metric collector, or HA architecture. Run the check manually after releases, or schedule it using the host scheduler once you have verified the behavior. Do not run it every second.

The sampler reads Compose service state and Docker's one-shot stats. It checks the proxied /ready path, reports absent or unhealthy services, and uses a nonzero exit code of 2 for actionable findings. Treat the 85% CPU/memory defaults as starting thresholds for a short local lab, not production SLOs. A missing memory percentage on Windows is skipped rather than treated as zero.

NEW TOOLS/OBSERVE_STACK.PY — COMPLETE FILE

```
"""Single-host status sampler. Read-only; returns 2 for actionable alerts.

Usage: python tools/observe_stack.py
Run from anywhere; script locates its own Compose project root.
"""
```

```
import argparse
import json
from pathlib import Path
import subprocess
import sys
from urllib.error import URLError, HTTPError
from urllib.request import urlopen

ROOT = Path(__file__).resolve().parents[1]
COMPOSE = ["docker", "compose", "-f", "compose.yaml",
          "-f", "compose.harden.yaml", "-f", "compose.observe.yaml"]
REQUIRED = {"db", "redis", "api", "publisher", "worker", "proxy"}


def decode_json_records(raw):
    """Accept Docker Compose JSON arrays or newline-delimited objects."""
    raw = raw.strip()
    if not raw:
        return []
    try:
        data = json.loads(raw)
        return data if isinstance(data, list) else [data]
    except json.JSONDecodeError:
        return [json.loads(line) for line in raw.splitlines() if line.strip()]


def run(*parts):
    outcome = subprocess.run([*parts], cwd=ROOT, capture_output=True,
                             text=True, timeout=30, check=False)
    if outcome.returncode:
        raise RuntimeError(f"Command exited {outcome.returncode}: {parts[0]} {parts[1]}")
    return outcome.stdout


def percentage(value):
    try:
        return float(str(value).strip().rstrip("%"))
    except (TypeError, ValueError):
        return None


def analyze(records, stats, cpu_limit=85.0, mem_limit=85.0):
    alerts = []
    by_service = {record.get("Service"): record for record in records}
    for service in sorted(REQUIRED):
        row = by_service.get(service)
        if not row:
            alerts.append(f"{service}: missing container")
```

```
            continue
        state = str(row.get("State", "")).lower()
        health = str(row.get("Health", "")).lower()
        if state != "running":
            alerts.append(f"{service}: state={state or 'unknown'}")
        if health == "unhealthy":
            alerts.append(f"{service}: health=unhealthy")
    for row in stats:
        # Container stats percentages are point-in-time signals, not trends.
        label = row.get("Name") or row.get("Container") or "container"
        cpu = percentage(row.get("CPUPerc"))
        mem = percentage(row.get("MemPerc"))
        if cpu is not None and cpu >= cpu_limit:
            alerts.append(f"{label}: CPU {cpu:.1f}%")
        if mem is not None and mem >= mem_limit:
            alerts.append(f"{label}: memory {mem:.1f}%")
    return alerts


def sample():
    records = decode_json_records(run(*COMPOSE, "ps", "--all", "--format", "json"))
    ids = [r.get("ID") for r in records if r.get("State") == "running" and r.get("ID")]
    stats = []
    if ids:
        stats = decode_json_records(run("docker", "stats", "--no-stream",
                                        "--format", "{{json .}}", *ids))
    return records, stats


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--cpu-warn", type=float, default=85.0)
    parser.add_argument("--mem-warn", type=float, default=85.0)
    args = parser.parse_args()
    try:
        records, stats = sample()
        alerts = analyze(records, stats, args.cpu_warn, args.mem_warn)
        for record in records:
            print(f"SERVICE {record.get('Service', '?')}: "
                  f"{record.get('State', '?')} / {record.get('Health', 'n/a')}")
        try:
            with urlopen("http://127.0.0.1:8080/ready", timeout=3) as response:
                if response.status != 200:
                    alerts.append(f"proxy /ready HTTP {response.status}")
        except HTTPError as exc:
            alerts.append(f"proxy /ready HTTP {exc.code}")
        except URLError:
            alerts.append("proxy /ready unreachable")
```

```
        if alerts:
            for alert in alerts:
                print("ALERT", alert)
            return 2
        print("OK one host sample; not a monitoring service or paging system")
        return 0
    except (OSError, RuntimeError, ValueError, json.JSONDecodeError,
subprocess.TimeoutExpired) as exc:
        print("ALERT cannot collect Docker evidence:", type(exc).__name__, file=sys.stderr)
        return 2


if __name__ == "__main__":
    raise SystemExit(main())
```

TERMINAL — SAMPLE AND INTERPRET

```
python tools/observe_stack.py
# Expected when healthy: all required services running;
# final line is OK one host sample ...
# Expected during a stopped Redis incident: ALERT ...
# Exit status: 0 = no alert; 2 = actionable alert / failed collection.
```

For a production host, replace this ad hoc sampler with centralized logs, retention policy, an external health observer, tested alert delivery, and escalation procedures. A monitor inside the same failed host cannot alert you when the host itself is down.

22.11 Checkpoint: Diagnose One Failure from Evidence

You will now deliberately interrupt Redis while leaving the API and PostgreSQL online. This is a development-only drill. It must not be attempted against production traffic. Redis has an append-only volume; the job ledger remains in PostgreSQL. The objective is to prove that the accepted job is not silently lost and that you can locate the failure from evidence alone.

A. Prepare the stack from the book

TERMINAL — REPEATABLE STARTUP (POWERSHELL OR BASH)

```
python tools/prepare_ch19_secrets.py
python tools/dc22.py config -q
python tools/dc22.py up -d --wait --wait-timeout 120 db redis
python tools/dc22.py build api
python tools/dc22.py run --rm migrate
```

```
python tools/dc22.py up -d --wait --wait-timeout 120 api publisher worker proxy
python tools/observe_stack.py
```

If you already have Chapter 21 running, keep the project name unchanged. The new logging override may cause container recreation, but it must not remove dbdata or redisdata. A successful sampler reports no alerts; if it does not, investigate the reported service before injecting any failure.

B. Create the test record and interrupt Redis

TERMINAL — ONE CONTROLLED DEPENDENCY OUTAGE

```
python tools/incident_probe.py prepare
python tools/dc22.py stop redis
python tools/incident_probe.py submit
python tools/incident_probe.py poll --seconds 5
python tools/observe_stack.py
python tools/dc22.py logs --since 3m --tail 50 redis publisher worker api
```

Expected: /ready can still return 200, the newly accepted job stays queued or otherwise unfinished, the sampler alerts on Redis not running, and publisher/worker logs show Redis connectivity errors. The precise retry messages depend on timing; absence of one individual log line does not disprove the outage. The API must not fabricate “succeeded” before a worker commits completion.

C. Restart, verify and clean up

TERMINAL — RECOVERY AND PROOF

```
python tools/dc22.py start redis
python tools/incident_probe.py poll --seconds 60 --require-succeeded
python tools/dc22.py logs --since 5m --tail 80 publisher worker
python tools/dc22.py run --rm migrate python tools/queue_snapshot.py
python tools/incident_probe.py cleanup
python tools/observe_stack.py
```

A successful run ends with a succeeded job and an otherwise healthy sampler. If the job does not complete, inspect pending Redis stream entries, worker logs, and durable attempts; do not delete the record or volume to hide a failure. Run the Chapter 21 quality gate again after changes. Persisted job IDs and timestamps belong in your incident worksheet.

Complete incident helper (no external download required)

The following entire source file is included so the book alone is sufficient to reproduce the request-and-recovery drill. Its state file contains IDs, not secrets, and is excluded from Git and image builds. Never send the generated state file or secret files to a public issue tracker.

NEW TOOLS/INCIDENT_PROBE.PY — COMPLETE FILE

```
"""Controlled localhost incident probe. Never echoes bearer tokens."""
import argparse
import json
from pathlib import Path
import time
from urllib.error import HTTPError, URLError
from urllib.request import Request, urlopen
from uuid import uuid4

ROOT = Path(__file__).resolve().parents[1]
STATE = ROOT / ".state" / "ch22_incident.json"
API = "http://127.0.0.1:8080"


def read_token():
    return (ROOT / "private" / "operator_token.txt").read_text(encoding="utf-8").rstrip("\r\n")


def request(method, path, body=None):
    payload = json.dumps(body).encode() if body is not None else None
    req = Request(API + path, data=payload, method=method,
                  headers={"Authorization": "Bearer " + read_token(),
                           "Content-Type": "application/json"})
    try:
        with urlopen(req, timeout=8) as response:
            data = response.read()
            rid = response.headers.get("X-Request-ID", "-")
            return response.status, json.loads(data) if data else {}, rid
    except HTTPError as exc:
        # Never print the exception or request headers (credentials).
        raise RuntimeError(f"API returned HTTP {exc.code}") from None
    except URLError:
        raise RuntimeError("API unreachable on localhost:8080") from None


def load():
    if not STATE.exists():
        raise RuntimeError("Run prepare first")
    return json.loads(STATE.read_text(encoding="utf-8"))


def save(state):
    STATE.parent.mkdir(exist_ok=True)
    STATE.write_text(json.dumps(state, indent=2) + "\n", encoding="utf-8")


def prepare():
```

```
    if STATE.exists():
        raise RuntimeError("Old incident state exists; inspect and cleanup first")
    label = "CH22-" + uuid4().hex[:10]
    status, obj, rid = request("POST", "/v1/deployments",
                               {"name": label, "environment": "staging", "version": "22.0"})
    if status != 201:
        raise RuntimeError(f"Unexpected create HTTP {status}")
    state = {"deployment_id": obj["id"], "request_key": "ch22-" + uuid4().hex,
             "job_id": None}
    save(state)
    print(f"PREPARED deployment={obj['id']} request_id={rid}")


def submit():
    state = load()
    status, obj, rid = request("POST", f"/v1/deployments/{state['deployment_id']}"
                                       f"/checks?key={state['request_key']}")
    if status != 202:
        raise RuntimeError(f"Unexpected submission HTTP {status}")
    state["job_id"] = obj["id"]
    save(state)
    print(f"SUBMITTED job={obj['id']} request_id={rid}")


def poll(limit, require_succeeded):
    state = load()
    if not state.get("job_id"):
        raise RuntimeError("Run submit first")
    deadline = time.monotonic() + limit
    while True:
        _, obj, rid = request("GET", f"/v1/checks/{state['job_id']}")
        status = obj["status"]
        print(f"JOB {state['job_id']} status={status} attempts={obj['attempts']} request_id={rid}")
        if status in ("succeeded", "failed") or time.monotonic() >= deadline:
            if require_succeeded and status != "succeeded":
                raise RuntimeError(f"Job did not succeed; status={status}")
            return
        time.sleep(2)


def cleanup():
    state = load()
    status, _, rid = request("DELETE", f"/v1/deployments/{state['deployment_id']}")
    if status != 204:
        raise RuntimeError(f"Unexpected deletion HTTP {status}")
    STATE.unlink()
    print(f"REMOVED lab record request_id={rid}")
```

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("action", choices=("prepare", "submit", "poll", "cleanup"))
    parser.add_argument("--seconds", type=int, default=60)
    parser.add_argument("--require-succeeded", action="store_true")
    args = parser.parse_args()
    try:
        if args.action == "prepare": prepare()
        elif args.action == "submit": submit()
        elif args.action == "poll": poll(args.seconds, args.require_succeeded)
        else: cleanup()
    except (OSError, ValueError, RuntimeError, KeyError) as exc:
        print("STOP:", exc)
        raise SystemExit(1)


if __name__ == "__main__":
    main()
```

ADD TO BOTH .GITIGNORE AND .DOCKERIGNORE

```
.state/
```

Focused tests for safe log output and health snapshots

These complete tests can run without Docker. Add the one new header test to tests/test_api.py as shown; the full updated test file is also present in the source archive.

NEW TESTS/TEST_OBSERVABILITY.PY — COMPLETE FILE

```
"""Fast isolated tests; does not require a Docker Engine."""
import json
import logging
from app.observability import JSONLogFormatter, request_id_from_header
from tools.observe_stack import decode_json_records, analyze, percentage


def test_request_id_accepts_safe_values_only():
    assert request_id_from_header("client_1234") == "client_1234"
    assert request_id_from_header("invalid\nheader") != "invalid\nheader"
    assert len(request_id_from_header("x")) == 32


def test_json_log_omits_unlisted_attributes():
    record = logging.LogRecord("deployboard.test", logging.INFO, __file__, 1,
                               "completed", (), None)
    record.event = "GET /health"
    record.status = 200
    record.authorization = "should-never-appear"
```

```
    data = json.loads(JSONLogFormatter().format(record))
    assert data["event"] == "GET /health"
    assert "authorization" not in data
    assert "should-never-appear" not in json.dumps(data)


def test_compose_ps_json_variants():
    rows = [{"Service": "api", "State": "running"}]
    assert decode_json_records(json.dumps(rows)) == rows
    assert decode_json_records(json.dumps(rows[0]) + "\n" + json.dumps(rows[0])) == rows * 2


def test_missing_or_unhealthy_service_alerts():
    rows = [{"Service": "api", "State": "running", "Health": "unhealthy"}]
    alerts = analyze(rows, [], cpu_limit=85, mem_limit=85)
    assert any("api: health=unhealthy" in a for a in alerts)
    assert any("db: missing" in a for a in alerts)


def test_percentages_and_thresholds():
    rows = [{"Service": s, "State": "running"} for s in
            ("api", "db", "redis", "publisher", "worker", "proxy")]
    assert analyze(rows, [{"Name": "api", "CPUPerc": "90.0%", "MemPerc": "6%"}]) == ["api:
CPU 90.0%"]
    assert percentage("unknown") is None
```

APPEND TO TESTS/TEST_API.PY

```
def test_request_trace_header_is_safe(client):
    response = client.get("/health", headers={"X-Request-ID": "audit-2026-22"})
    assert response.headers["X-Request-ID"] == "audit-2026-22"
    invalid = client.get("/health", headers={"X-Request-ID": "tiny"})
    assert invalid.headers["X-Request-ID"] != "tiny"
    assert len(invalid.headers["X-Request-ID"]) == 32
```

Local regression and evidence checklist

TERMINAL — QUALITY CHECKS

```
python -m pytest -q tests/test_observability.py tests/test_api.py
python -m compileall -q app tools tests
python tools/dc22.py config -q
# Full integration regression (Chapter 21):
python tools/quality_gate.py
```

The Chapter 21 quality gate runs in its own deployboard-qa21 Compose project; the observability incident stays within deployboard-ch14. The gate checks baseline functionality, while this chapter adds evidence and safe diagnosis. Do not confuse the two project volumes.

RELEASE EVIDENCE	ACCEPTANCE RULE
Health and readiness	/health and /ready serve expected statuses; Redis outage interpretation documented
Correlation	HTTP response ID matches an API log; job ID matches persisted status and worker outcome
Retention	local driver, 10m/3 limits visible in effective Compose config; recreated containers use settings
Resource sampler	Healthy exit 0; stopped Redis produces actionable alert / exit 2
Recovery	One check queued during outage completes after Redis restoration; no volume deletion
Regression	Chapter 21 gate and any updated worker tests pass in a live Docker environment

Release decision: PASS only when each acceptance rule has actual recorded evidence. HOLD if a job is stuck, logs expose a credential, a service is unhealthy, or any integrated test fails. This chapter improves observability; it does not add automatic failover, metrics history, or a multi-host alert pipeline.

CHAPTER 23

Troubleshoot Startup, Network, and Storage Failures

A container that is “running” can still be unable to serve a request. The operator’s job is not to restart it repeatedly; it is to identify the failing boundary, change one thing, and prove the system has recovered.

THE OPERATOR’S PROMISE

You will be able to…	Evidence you will keep
Distinguish process, readiness, network, and storage failures	Status, exit reason, health history, port and route map
Repair Compose startup and configuration errors safely	Rendered configuration and before/after checks
Resolve an incident without harming the live database	Isolated lab, precise remediation, explicit data boundary
Close the incident only after verification	Redacted receipt with repeatable commands and results

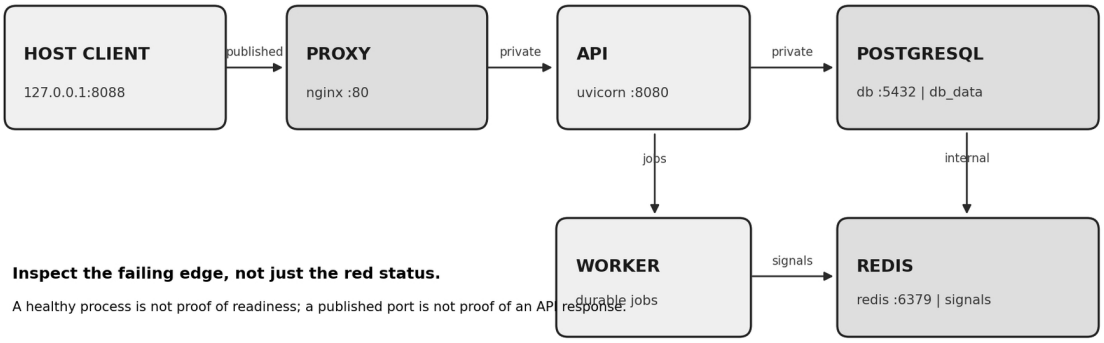


Figure 23.1 Read DeployBoard as a chain of observable boundaries. Test the failing edge before touching the next one.

SCOPE AND SAFETY The book’s Appendix E is the canonical six-service DeployBoard kit. The lab in §23.11 uses a new Compose project, locally generated secrets, and a disposable working copy. Do not run destructive cleanup on a live system. No Docker Engine is available in the

manuscript-build environment, so live failure outcomes are instructions to verify, not claims of executed integration tests.

23.1 Use a Consistent Observe–Hypothesize–Test Workflow

Production debugging becomes reliable when every change answers a precise question. The workflow is OBSERVE → LOCATE → HYPOTHESIZE → TEST → VERIFY → RECORD. The order matters: collect state before a restart changes it.

Step	Operator question	Minimum evidence
Observe	What is actually failing, and since when?	Incident time (UTC), URL, command, exit code
Locate	Which boundary first differs from expectation?	Compose state, service logs, port mapping
Hypothesize	What single cause explains those observations?	One falsifiable claim and predicted test
Test	What reversible action distinguishes causes?	Command, owner, risk, result
Verify	Is the user path and stored data intact?	HTTP result, role check, DB/worker checks
Record	Could another operator repeat this?	Redacted receipt and follow-up

INITIAL READ-ONLY INSPECTION, FROM THE PROJECT DIRECTORY

```
docker compose ps --all
docker compose logs --no-color --tail=80 api proxy db
docker compose config --quiet
docker compose port proxy 80
```

Do not treat a blank error page as proof that the API failed. The request may have been refused by the host, terminated by the proxy, blocked by a dependency, or rejected by authentication. Record the first failing hop and inspect outward only if that hop is sound.

DECISION RULE Before editing an image, YAML file, volume, or firewall rule, write: “If X is the cause, command Y should produce signal Z.” A result that contradicts the hypothesis is useful evidence, not a reason to try a random second change.

23.2 Diagnose Crash Loops and Startup Races

Begin with lifecycle state. Exited means the main process ended. Restarting indicates a policy is bringing it back. Running with an unhealthy check means the process exists but the defined health command is failing. A one-shot migration that exits 0 is often correct; it is not a crash loop.

CAPTURE THE API STATE (BASH / MACOS / WSL)

```
docker compose ps --all
docker compose logs --no-color --tail=100 migrate api
CID=$(docker compose ps -q api)
[ -n "$CID" ] && docker inspect -f '{{.State.Status}} exit={{.State.ExitCode}} oom={{.State.OOMKilled}}' "$CID"
```

Observation	Likely next probe	Do not assume
migrate: exited (0)	Check schema_migrations and API startup	Every exited container is broken
api: restarting	Inspect first fatal exception and exit code	A higher restart count will fix a bad config
api: running, unhealthy	Call /readyz in container and trace DB connection	The host port is necessarily wrong
proxy not started	Inspect api health and depends_on conditions	Nginx configuration is necessarily invalid

DeployBoard's api waits for migrate to complete successfully and Redis to be healthy; the proxy waits for the api healthcheck. The database also has a readiness probe. Compose can order services and wait on health conditions, but an application must still handle dependencies disappearing after startup.

ONLY AFTER CORRECTING A SPECIFIC API CONFIG CAUSE

```
docker compose logs --no-color --tail=60 db migrate redis
docker compose config --quiet
docker compose up -d --no-deps api
```

AVOID A FALSE FIX Recreating the API may briefly make the health indicator green while leaving the original trigger intact. Verify the same operation after at least one subsequent probe interval, and check a real authorized API path as well.

23.3 Trace DNS and Connection Refusal Problems

DNS resolution, TCP connection, and application response are different tests. A service name such as db is meaningful on the Docker backend network; localhost inside the api container refers to the api container itself, not the PostgreSQL container or the host.

NAME RESOLUTION, THEN TCP, FROM A RUNNING API CONTAINER

```
docker compose exec api python -c "import socket; print(socket.getaddrinfo('db',5432)[0][4])"
docker compose exec api python -c "import socket; s=socket.create_connection(('db',5432),3); print('TCP OK'); s.close()"
```

Probe result	Interpretation	Next evidence
Name does not resolve	Wrong DB_HOST, service name, or network membership	docker compose config; network inspect
Name resolves; connection refused	Target reached but nothing listening on selected port	db logs, service health, DB_PORT
TCP succeeds; API /readyz fails	Likely database credentials, schema, or SQL error	redacted API logs and DB readiness
Works inside; fails from host	Published port, loopback binding, or proxy path	docker compose port proxy 80; curl -i

For a service that is too unhealthy for docker compose exec, examine its logs first. A temporary docker compose run --rm --no-deps api python -c ... process can be useful, but it is a new container with different startup behavior; record that limitation before interpreting its result.

SECURITY BOUNDARY The backend network is internal. Do not publish PostgreSQL or Redis ports to make a probe easier; run diagnostics from a permitted container or an authorized maintenance environment.

23.4 Locate Wrong Host and Container Ports

DeployBoard's proxy listens on port 80 inside its container; the host publishes 127.0.0.1:${HTTP_PORT:-8088} to it. The API listens on internal port 8080. Nothing in the canonical file publishes the db or redis service to the host. These are intentionally different addresses.

HOST CHECK WHEN HTTP_PORT USES ITS DEFAULT

```
docker compose port proxy 80
curl -i http://127.0.0.1:8088/healthz
curl -i http://127.0.0.1:8088/readyz
```

Address	Meaning	Expected use
127.0.0.1:8088 on host	Loopback-bound published proxy port (default)	Local browser and curl
proxy:80 inside backend	Nginx listening inside its container	Docker internal hop
api:8080 inside backend	FastAPI process listener	Nginx upstream and internal checks
db:5432 inside backend	PostgreSQL connection endpoint	Application and migration only

If a command returns "connection refused," confirm the actual binding before investigating HTTP authentication. If it returns 401 or 403, transport

succeeded and the problem lies in application policy or supplied credentials. If it returns 502 or 504, the proxy received your request but could not complete the upstream exchange.

PLATFORM LIMIT Docker Desktop places Linux containers behind a VM. Host port behavior and local firewall inspection differ on macOS and Windows. The canonical default is local-only; public HTTPS requires a separately verified and secured ingress.

23.5 Investigate Volume Ownership and Mount Errors

Persistent state has a different repair cost from an application container. DeployBoard stores PostgreSQL data in the db_data named volume; Nginx receives a read-only bind-mounted configuration file. A mistaken path in the proxy bind mount and a PostgreSQL data-directory permission error may both prevent startup, but they require different actions.

OBSERVE WITHOUT EDITING STORAGE

```
docker compose ps --all
docker compose logs --no-color --tail=80 db proxy
docker volume ls
docker compose exec db sh -c 'id; ls -ld /var/lib/postgresql/data'
```

Symptom	Low-risk check	Safe repair principle
Nginx complains that config path is missing	Verify nginx/default.conf exists in project copy	Restore file path; do not loosen all host perms
Postgres reports permission denied	Inspect mount source, uid/gid, filesystem type	Match owner to image requirements after backup
Volume seems empty after new -p name	Inspect Compose project and volume name	Do not point a test at a live data volume
Expected data vanishes after recreate	Confirm correct project, mount name, and backup	Restore from a verified backup only

Never respond to a data-directory error with docker compose down -v, recursive chmod 777, or an unreviewed chown -R. Such commands can destroy data or conceal the true owner mismatch. For a bind mount, inspect host ownership and ACLs; for a named volume, identify the volume and backup before any permission change.

EVIDENCE THRESHOLD A running database is not proof of persistence. Confirm a known record exists after a controlled container recreate and

preserve the before/after query result. Restore drills belong in an isolated database as shown in Appendix I.

23.6 Identify Disk Exhaustion and Broken Log Rotation

Disk incidents can masquerade as database corruption, build failure, or a container that refuses to start. Separate bytes exhausted, inodes exhausted, and oversized container logs; each requires a different resolution. Docker Desktop also has VM disk-capacity considerations that host-only df output may not reveal.

OBSERVE STORAGE AND LOGGING ON A LINUX DOCKER HOST

```
docker system df -v
df -h
df -i
docker info --format '{{.LoggingDriver}}'
CID=$(docker compose ps -q api)
[ -n "$CID" ] && docker inspect -f '{{.HostConfig.LogConfig.Type}}' "$CID"
```

A default unrotated json-file logging setup may fill the Docker data root; other drivers have different behavior. Logging configuration must be evaluated at both daemon and container scope. On a disposable lab service, a Compose logging configuration can make retention explicit:

ILLUSTRATIVE COMPOSE FRAGMENT; MERGE UNDER A SERVICE

```
services:
  api:
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "3"
```

Rotate and retain logs in a way that preserves your organization’s audit requirements. Do not truncate files under Docker’s internal data directory by hand. Target only artifacts that have been reviewed as removable; broad system prune and volume prune commands are deliberately excluded from this repair procedure.

23.7 Understand Out-of-Memory Exits and Resource Limits

A container exit code of 137 commonly reflects SIGKILL, but SIGKILL is not synonymous with out-of-memory. Inspect the OOMKilled flag and host

or platform memory pressure. A Linux host's cgroups and a Docker Desktop VM enforce limits differently; a one-off spike is not enough to choose a permanent limit.

CHECK MEMORY BEFORE INCREASING A LIMIT

```
docker stats --no-stream
CID=$(docker compose ps -q api)
[ -n "$CID" ] && docker inspect -f 'exit={{.State.ExitCode}} oom={{.State.OOMKilled}}' "$CID"
```

Evidence	Interpretation	Candidate response
Exit 137; OOMKilled=true	Kernel or cgroup kill is strongly indicated	Measure peak, request sizes, worker concurrency
Exit 137; OOMKilled=false	SIGKILL also possible without OOM	Inspect deployment action and host logs
Memory rises over hours	Possible leak or unbounded workload	Reproduce under a measured load test
CPU high, memory stable	Slow path is not necessarily an OOM problem	Profile request or worker throughput

Resource limits are safeguards, not performance cures. Increase them only when measurements and host capacity justify the change. First examine unbounded queues, retries, large request payloads, concurrent reports, and what the worker does when a job fails.

23.8 Debug Configuration and Environment Precedence

Compose interpolation chooses values used while rendering the configuration. Container environment injection determines values actually passed to the process. A host shell export can change ${HTTP_PORT} without adding any variable to the api container; a mounted secret file may not appear in environment variable output at all.

VERIFY INTERPOLATION (REDACT BEFORE SAVING OR SHARING)

```
docker compose config --quiet
docker compose config --environment
docker compose port proxy 80
```

For the canonical kit, DB_HOST=db and DB_PORT=5432 are container variables, while HTTP_PORT is read when Compose renders the proxy's published port. In a development workstation, .env and the shell may supply different values. Compose's precedence rules decide the final result; printing the final model is more reliable than reading one file and assuming it wins.

Unexpected value	Check first	Likely correction
Host proxy port is not 8088	HTTP_PORT shell/.env; actual compose port output	Use the published port or correct the variable
API contacts missing-db	Final api environment in rendered Compose model	Remove the offending overlay
Secret-file error	File exists; filename and read permission	Regenerate lab credentials; protect host directory
Wrong project picked up	-p flag, COMPOSE_PROJECT_NAME, current directory	Declare one explicit training project name

REDACTION docker compose config, docker inspect, docker compose logs and process dumps can expose values or paths that should not be published. Do not paste raw production output into a book, ticket, or public discussion. Record only the necessary redacted findings.

23.9 Repair Failed Builds and Stale Cache Assumptions

A build error and a runtime error demand different tests. First prove that the Dockerfile can see the required files in its build context and that .dockerignore is not excluding them. Then distinguish a stale cached RUN layer from a pin that legitimately resolves to a changed image tag.

BUILD AND INSPECT BEFORE FORCING A COLD REBUILD

```
docker compose config --quiet
docker compose build --progress=plain api
docker image ls
docker compose logs --no-color --tail=60 api
```

If a cache-specific hypothesis survives inspection, a diagnostic rebuild without cache is appropriate on a training project:

TARGETED CACHE TEST; NETWORK AND REGISTRY ACCESS REQUIRED

```
docker compose build --pull --no-cache api
```

BuildKit cache invalidation does not depend on file modification time alone, and RUN instructions do not automatically refresh external packages merely because time passes. Do not prune all builders to repair a single bad input. Preserve the failing command, the corrected source, the chosen base-image digest, and the successful result.

23.10 Capture a Reproducible Incident Record

An incident report is useful when it answers four questions: What did the user experience? What observations ruled out alternatives? Which reversible change was made? What proved recovery without harming data? Use UTC timestamps, the explicit Compose project name, and the exact command syntax. Do not include credentials or unredacted application content.

Field	What to record
Impact + start	Which environment, user-facing symptom, first detected UTC time
Evidence	Command, UTC time, redacted finding, observed exit/status
Hypothesis	One falsifiable cause; include rejected hypotheses
Repair	Approved change, owner, scope, before/after state
Verification	Liveness, readiness, authorized operation, data and worker behavior
Rollback	How to reverse the change without touching persistent state
Close	Recorded time and follow-up preventive control

The companion lab contains a tiny offline receipt validator. It checks required fields, timezone-aware evidence timestamps, and proof required before marking an incident resolved. It never contacts Docker or asserts that a fix worked. The listing below is complete; save it beside incident_template.json.

LISTING 23.1 LAB/INCIDENT_CHECK.PY — OFFLINE RECEIPT CHECKER [LINES 1–27]

```
"""Offline sanity checker for a redacted DeployBoard incident receipt.

Usage: python incident_check.py incident_template.json
It checks structure and closure evidence; it cannot verify Docker behavior.
"""
import json
import sys
from datetime import datetime
from pathlib import Path

REQUIRED = ("incident_id", "project", "status", "started_utc", "impact",
            "symptom", "evidence", "hypothesis", "safe_change",
            "verification", "rollback_plan", "closed_utc")
STATES = {"open", "monitoring", "resolved"}


def timestamp(value):
    if not isinstance(value, str) or not value:
        return False
    try:
        parsed = datetime.fromisoformat(value.replace("Z", "+00:00"))
        return parsed.utcoffset() is not None
    except ValueError:
        return False


def validate(data):
```

LISTING 23.1 LAB/INCIDENT_CHECK.PY — OFFLINE RECEIPT CHECKER [LINES 28–54]

```
    errors = []
    if not isinstance(data, dict):
        return ["Receipt must be a JSON object"]
    for field in REQUIRED:
        if field not in data:
            errors.append(f"Missing: {field}")
    if errors:
        return errors
    for field in ("incident_id", "project", "impact", "symptom", "rollback_plan"):
        if not isinstance(data[field], str) or not data[field].strip():
            errors.append(f"Required text: {field}")
    if not isinstance(data["status"], str) or data["status"] not in STATES:
        errors.append("Invalid status")
    if not timestamp(data["started_utc"]):
        errors.append("started_utc requires a timezone-aware timestamp")
    if not isinstance(data["evidence"], list) or not data["evidence"]:
        errors.append("At least one evidence record is required")
    else:
        for position, item in enumerate(data["evidence"], 1):
            if not isinstance(item, dict):
                errors.append(f"Evidence {position}: not an object")
                continue
            if not timestamp(item.get("time_utc")):
                errors.append(f"Evidence {position}: missing UTC time")
            for field in ("command", "finding"):
                if not isinstance(item.get(field), str) or not item[field].strip():
                    errors.append(f"Evidence {position}: missing {field}")
```

LISTING 23.1 LAB/INCIDENT_CHECK.PY — OFFLINE RECEIPT CHECKER [LINES 55–81]

```
    if data["status"] == "resolved":
        for field in ("hypothesis", "safe_change", "verification"):
            if not isinstance(data[field], str) or not data[field].strip():
                errors.append(f"Resolved receipt missing: {field}")
        if not timestamp(data["closed_utc"]):
            errors.append("Resolved receipt requires closed_utc")
        elif timestamp(data["started_utc"]):
            start = datetime.fromisoformat(data["started_utc"].replace("Z", "+00:00"))
            end = datetime.fromisoformat(data["closed_utc"].replace("Z", "+00:00"))
            if end < start:
                errors.append("closed_utc precedes started_utc")
    return errors

def main(argv):
    if len(argv) != 2:
        print("Usage: python incident_check.py RECEIPT.json", file=sys.stderr)
        return 2
    try:
        data = json.loads(Path(argv[1]).read_text(encoding="utf-8"))
    except (OSError, json.JSONDecodeError) as exc:
        print(f"Receipt could not be opened: {exc}", file=sys.stderr)
        return 2
    errors = validate(data)
    if errors:
        for error in errors:
            print("FAIL:", error)
```

LISTING 23.1 LAB/INCIDENT_CHECK.PY — OFFLINE RECEIPT CHECKER [LINES 82–88]

```
        return 1
    print("PASS: receipt structure complete; live behavior not verified")
    return 0

if __name__ == "__main__":
    raise SystemExit(main(sys.argv))
```

Create the following starter receipt. It deliberately starts as open; leaving verification blank is correct before the experiment.

LISTING 23.2 LAB/INCIDENT_TEMPLATE.JSON — EDITABLE INCIDENT RECORD

```
{
  "incident_id": "DB23-001",
  "project": "deployboard23lab",
  "status": "open",
  "started_utc": "2026-10-21T00:00:00Z",
  "impact": "Local training stack only; no customer traffic",
  "symptom": "API unhealthy; proxy not ready",
  "evidence": [
    {
      "time_utc": "2026-10-21T00:02:00Z",
      "command": "docker compose -p deployboard23lab ps --all",
      "finding": "API unhealthy; inspect dependency chain"
    }
  ],
  "hypothesis": "",
  "safe_change": "",
  "verification": "",
  "rollback_plan": "Keep existing named volume; remove lab overlay only",
  "closed_utc": ""
}
```

RUN THE OFFLINE RECEIPT CHECK

```
python lab/incident_check.py lab/incident_template.json
# Expected: PASS: receipt structure complete; live behavior not verified
```

23.11 Checkpoint: Resolve a Multi-Service Failure Lab

This scenario plants two separable defects: a false database hostname in the API service and a client assumption that the host proxy is always on port 8088. The worker and PostgreSQL are intentionally not broken. Use an isolated Compose project and discover each fault from evidence rather than guessing.

LAB CONTRACT

Required	Boundary
Working source	A NEW copy of the complete Appendix E DeployBoard_Complete_Kit
Lab identity	Always pass -p deployboard23lab; never reuse the live project
Environment	Bash shell on Linux, macOS, or WSL2; Docker Engine and Compose v2.20+
Credentials	Generate NEW lab secrets; do not copy production tokens
Protection	No public bind; no down -v, prune --volumes, or schema reset

Build — stage a safe, falsifiable failure

Begin from a clean, copied Appendix E kit, not the production directory. From the parent of DeployBoard_Complete_Kit, use the following commands (or start in the separate lab folder provided in the ZIP). If your host already uses 18088, choose another free loopback port and change the curl commands accordingly.

COPY INTO AN ISOLATED PRACTICE DIRECTORY (BASH / MACOS / WSL)

```
cp -R DeployBoard_Complete_Kit DeployBoard_Chapter23_Lab
cd DeployBoard_Chapter23_Lab
```

Copy .env.example to .env and generate NEW local secrets. The supplied bad-host overlay is a complete file and is printed here:

LISTING 23.3 LABS/CH23_BAD_API.YAML — INTENTIONAL WRONG HOSTNAME

```
# Intentional training failure. Run only with a distinct -p project name.
# Never combine this overlay with a real production project.
services:
  api:
    environment:
      DB_HOST: missing-db
```

EXECUTE ONLY IN THE COPIED, PRIVATE TRAINING DIRECTORY

```
cp .env.example .env
python scripts/generate_secrets.py
mkdir -p labs
# Save Listing 23.3 as labs/ch23_bad_api.yaml
export HTTP_PORT=18088
docker compose -p deployboard23lab -f compose.yaml -f labs/ch23_bad_api.yaml config --quiet
docker compose -p deployboard23lab -f compose.yaml -f labs/ch23_bad_api.yaml up -d --build
```

EXPECTED FAILURE Compose up may return a nonzero exit code because the API never becomes healthy and the proxy depends on API health. That is part of the exercise, not a reason to delete a volume. Some services will still run; capture their state before changing anything.

Verify — locate the first failing boundary

COMPARE THE FAILING HOSTNAME TO THE HEALTHY DATABASE SERVICE

```
docker compose -p deployboard23lab ps --all
docker compose -p deployboard23lab logs --no-color --tail=80 api migrate db
docker compose -p deployboard23lab -f compose.yaml -f labs/ch23_bad_api.yaml config
docker compose -p deployboard23lab port proxy 80
```

Your diagnostic hypothesis should say: “PostgreSQL may be healthy, but the API is configured to contact missing-db, so its readiness check fails; the dependent proxy may be unable to start.” A separate client probe to 127.0.0.1:8088 may fail because this lab deliberately publishes a different host port. Neither finding justifies altering db_data.

Break and repair — remove only the bad overlay

REMOVE THE WRONG DB_HOST; USE THE ACTUAL 18088 PUBLISHED HOST PORT

```
docker compose -p deployboard23lab -f compose.yaml up -d --build
docker compose -p deployboard23lab ps --all
docker compose -p deployboard23lab port proxy 80
curl -i http://127.0.0.1:18088/healthz
curl -i http://127.0.0.1:18088/readyz
```

A successful health request should return HTTP 200 with an alive status; readiness should return HTTP 200 only if its database query succeeds. Results vary if images cannot be downloaded or dependencies fail. If the proxy remains blocked, inspect status and logs before rerunning. The wrong

URL 127.0.0.1:8088 is not a reliable lab probe: a separate local application might already own it.

Ship — prove and document recovery

Complete the receipt by recording the wrong hostname, the Compose project and port map, the change made, and the successful live probes. Confirm an authorized GET /v1/tasks with the viewer token through the proxy and a real worker job after the stack is healthy. Do not copy the token into your incident record. The worker has no Compose healthcheck in the canonical kit, so status “running” alone does not prove queue progress.

RECORD EVIDENCE FIRST, THEN STOP LAB CONTAINERS WHILE RETAINING VOLUMES

```
python lab/incident_check.py lab/incident_template.json
docker compose -p deployboard23lab down
# NEVER append -v to this training cleanup command.
```

Acceptance test	Pass evidence / FAIL rule
Isolation	-p deployboard23lab visible on every Compose action
Root cause	Bad DB_HOST evidenced by rendered config + API readiness failure
Service recovery	API and proxy start; /healthz and /readyz respond as expected
Network boundary	Only proxy bound to loopback; db and redis remain private
Data protection	No volume deletion; persisted test record retained if one was created
Incident report	Time-stamped redacted evidence, controlled change, verification

If any acceptance test fails, the checkpoint is incomplete. Return to the observation that failed and repeat one hypothesis-driven check. Avoid presenting sample outputs as though you observed them on your own machine.

FIELD GUIDE / SYMPTOM-TO-FIRST-COMMAND INDEX

Symptom	First question / command	Core section
Restarting	Why did PID 1 exit? docker compose logs --tail=80 api	23.2
Unhealthy	Is a dependency actually ready? /readyz and logs	23.2, 23.3
Name cannot resolve	Which Docker network and service name?	23.3
Connection refused	Which port, listener, and namespace?	23.3, 23.4
Permission denied	What is mounted, who owns it?	23.5
No space left on device	Bytes or inodes? docker system df -v	23.6
Exited 137	OOMKilled true? docker stats --no-stream	23.7
Unexpected env value	Final Compose model and interpolation source?	23.8
Outdated image	Cache behavior, context, and source digest?	23.9
“Fixed” but no proof	Which acceptance tests and record?	23.10, 23.11

Chapter 24 — Optimize Build, Runtime, and Operational Performance

A Docker release is not faster because its image is smaller, and it is not reliable because a benchmark prints an impressive number. You need evidence that the application preserves its contract while consuming fewer build minutes, fewer resources, or less operational effort. This chapter turns the already secured and observable DeployBoard stack into a controlled optimization experiment.

What you will deliver

A baseline/candidate comparison for image builds, authenticated API latency, background-job throughput, resource use, and likely registry/storage expense; an auditable release decision; and a repeatable performance lab that never touches the normal DeployBoard database volume.

Your starting point

Use the complete DeployBoard files carried forward from Chapter 23, Docker Engine with Compose V2 and Buildx, and Python 3.11 or newer on your host. The Chapter 24 ZIP includes the cumulative source plus every new file printed below. Work in that directory. Have at least 2 GiB of free host RAM and several GiB of free disk space before running containers. Adjust only after measuring your machine.

Safety boundary

The helper targets project deployboard-perf24 and host address 127.0.0.1:18024. The two named database/Redis volumes belong to that isolated project. Never substitute your main production Compose project or use volume-deleting cleanup as part of a speed test. Do not run load tests against a public site or an endpoint you do not control.

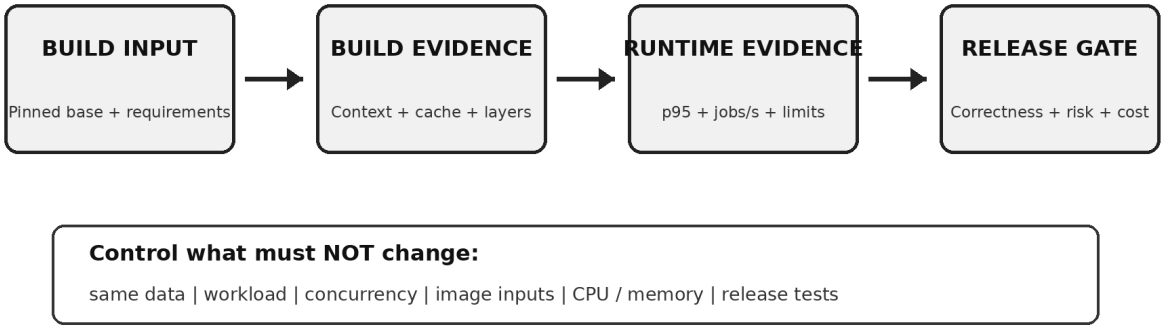


Figure 24.1 | Measure the whole delivery path, not just image size.

24.1 Establish Baseline Build and Runtime Measurements

Define the workload and its operating envelope before changing Dockerfiles. A build measurement includes the builder and cache state; an HTTP measurement includes network, application, and database work; a job measurement includes Redis delivery and PostgreSQL durability. None of these metrics is interchangeable.

Signal	Collect	Keep constant
Build duration	Wall-clock elapsed time for each build	Same builder, platform, Dockerfile inputs and network
Image size	docker image inspect .Size (bytes)	Compare runtime images, not cache directories
API p50/p95, errors	40 GETs, concurrency 4, warmed up	Same 20 production records and service limits
Worker jobs/s, failures	10 new asynchronous checks per run	Same worker count, retry policy, no backlog
Memory/CPU/restarts	docker stats; docker inspect	Same CPU and memory budgets; sustained observation

Make a measurable record

```
docker version
docker compose version
docker buildx version
docker buildx ls
docker system df
```

```
# Read-only inventory: never run prune to create a favorable result.
```

Capture your Docker/Compose versions, CPU architecture, available host memory, base-image reference, pinned requirements, and the timestamp of each run. Take at least three repeated measurements where time permits, and compare medians. In this chapter the first quick run is a learning checkpoint, not a claim of statistically conclusive performance.

24.2 Reduce Build Context and Unused Image Layers

Docker sends a filtered build context to BuildKit. Excess downloads, test reports, caches, secret files, and local data waste I/O and may expose files to the build. Preserve the existing Chapter 19 exclusions and add only the new performance evidence patterns shown here. Excluding build-context content does not delete it from the working tree.

Append to the existing .dockerignore

```
# Chapter 24 generated performance artifacts
evidence/ch24/
.coverage
*.prof
```

The original Dockerfile already copies requirements ahead of application code, and runs as the dedicated app user. Review it instead of applying generic advice such as “add more RUN layers” or “use Alpine.” Your actual context and final image contents determine whether any structural reduction helps.

```
docker buildx du
docker image ls deployboard --format "table {{.Repository}}:{{.Tag}}\t{{.Size}}"
# After the baseline build:
docker image inspect deployboard:ch24-baseline --format "{{.Size}}"
```

24.3 Compare Dependency and Base-Image Tradeoffs

The existing runtime uses python:3.13-slim and pinned Python package versions. It is a reasonable controlled reference: compatible wheels, a predictable Debian-based userspace, and no compiler in the finished image. Distroless or Alpine may lower some image sizes, but musl/glibc

compatibility, debugging cost, vulnerability availability, and startup behavior differ. A smaller image is not automatically faster or safer.

Candidate decision record

Option	Potential gain	What could disqualify it
Stay on python:3.13-slim	Reproducible behavior; low change risk	No measured benefit; patch cadence still required
Alternative slim image digest	Security update without structural change	Different runtime/dependency behavior
Alpine or minimal runtime	Potentially smaller compressed transfer	Wheels, libc, runtime tools, diagnosis costs
Multi-stage dependency build	Remove compiler/build-only tools	Extra complexity when binary wheels already suffice

Keep the same base and requirements for the primary experiment. If you change those inputs later, call that a separate experiment and rerun Chapter 20's image scan, SBOM review, and Chapter 21's quality gate.

24.4 Tune Caching Without Compromising Correctness

BuildKit caches layers whose instructions and inputs still match. A persistent pip cache mount can reduce repeated downloads after a requirements change without turning package downloads into final image layers. It does not promise a smaller final image or a faster first build. Do not cache the contents of secret mounts and never replace dependency pinning with unbounded "latest" versions.

SOURCE FILE / Dockerfile.optimized

```
# syntax=docker/dockerfile:1
# Chapter 24 candidate: identical application/runtime dependencies; cache pip downloads.
FROM python:3.13-slim
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1
WORKDIR /service
COPY requirements.txt .
RUN --mount=type=cache,target=/root/.cache/pip \
    pip install -r requirements.txt
RUN groupadd --system app && \
    useradd --system --gid app --no-create-home app
COPY --chown=app:app app ./app
COPY --chown=app:app migrations ./migrations
COPY --chown=app:app alembic.ini ./alembic.ini
COPY --chown=app:app tools ./tools
```

```
USER app
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

The candidate deliberately retains the same OS image family, pinned requirements, application paths, user, and entry point. The only meaningful experiment is the BuildKit pip cache mount. Because a warm builder may already have cached the full dependency layer, change/no-change rebuilds are different trials and should be labeled separately.

Use a timing wrapper to preserve raw build evidence

SOURCE FILE / tools/time24.py

```
"""Time one build command; preserve its exit status and aggregate evidence."""
import argparse
import json
import subprocess
import time
from pathlib import Path


def measure(command):
    started = time.perf_counter()
    code = subprocess.call(command)
    return {"elapsed_s": round(time.perf_counter() - started, 3),
            "exit_code": code}


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("--label", choices=["baseline-first", "baseline-repeat",
                        "candidate-first", "candidate-repeat"], required=True)
    parser.add_argument("command", nargs=argparse.REMAINDER)
    args = parser.parse_args()
    command = args.command[1:] if args.command[:1] == ["--"] else args.command
    if not command:
        parser.error("Supply the build command after --")
    result = measure(command)
    result["label"] = args.label
    directory = Path("evidence/ch24")
    directory.mkdir(parents=True, exist_ok=True)
    dest = directory / (args.label + "-build.json")
    dest.write_text(json.dumps(result, indent=2) + "\n", encoding="utf-8")
    print("Build result:", json.dumps(result), "saved to", dest)
    raise SystemExit(result["exit_code"])


if __name__ == "__main__":
```

```
    main()
```

```
python tools/time24.py --label baseline-first -- python tools/dc24.py --variant baseline build api
python tools/time24.py --label baseline-repeat -- python tools/dc24.py --variant baseline build api
python tools/time24.py --label candidate-first -- python tools/dc24.py --variant candidate build api
python tools/time24.py --label candidate-repeat -- python tools/dc24.py --variant candidate build
api
```

Time each command with the clock of your own shell, recording the first and repeat runs independently. Existing caches can favor either side; do not describe a repeat build as a "cold build." For stronger comparisons, use fresh dedicated builders and identical cached inputs, not system-wide pruning. Inspect the two image sizes before claiming an image-size improvement.

24.5 Set Realistic Container CPU and Memory Limits

CPU and memory limits are safety rails, not automatic performance optimizations. An undersized worker can stall the queue; an undersized API can push p95 latency upward; a too-small database limit can cause OOM kills and recovery delays. Start with the Chapter 18 budgets and hold them constant between both image variants.

SOURCE FILE / compose.perf.yaml

```
# Additive to compose.yaml, compose.harden.yaml, compose.observe.yaml.
# Keep the same fixed runtime budgets in BOTH measured variants.
services:
  api:
    image: ${DEPLOYBOARD_IMAGE:-deployboard:ch24-baseline}
    build:
      context: .
      dockerfile: ${DEPLOYBOARD_DOCKERFILE:-Dockerfile}
    mem_limit: 384m
    cpus: 1.0
  publisher:
    image: ${DEPLOYBOARD_IMAGE:-deployboard:ch24-baseline}
  worker:
    image: ${DEPLOYBOARD_IMAGE:-deployboard:ch24-baseline}
    mem_limit: 256m
    cpus: 0.5
  migrate:
    image: ${DEPLOYBOARD_IMAGE:-deployboard:ch24-baseline}
```

Both variants use the same 384 MiB API memory ceiling and 1 CPU; the worker uses 256 MiB and half a CPU. PostgreSQL and Redis retain their existing Compose settings. If the host is small, reduce request concurrency first rather than cutting memory arbitrarily. Re-run correctness tests after any subsequent resource change.

Inspect the effect, not just the YAML

```
python tools/dc24.py --variant baseline config --quiet
python tools/dc24.py --variant baseline ps
docker stats --no-stream
# Use docker inspect on a specific container to confirm applied limits.
# For OOM investigations: docker inspect <container> --format "{{.State.OOMKilled}}"
```

24.6 Measure API Latency and Worker Throughput

The baseline uses an authenticated HTTP request through the localhost Caddy proxy. Production records form a stable 20-row test fixture, while job experiments create records only in the staging environment. API samples therefore do not grow merely because a worker experiment ran. The test includes five warm-up requests, bounded parallelism, a wall-clock throughput measure, nearest-rank p50/p95, and a nonzero exit on failures.

Run everything in a separate Compose project

SOURCE FILE / tools/dc24.py

```
"""Run Chapter 24 only in the isolated deployboard-perf24 project."""
import argparse
import os
import subprocess

FILES = ["compose.yaml", "compose.harden.yaml",
         "compose.observe.yaml", "compose.perf.yaml"]


def make_command(variant, command):
    if not command:
        raise ValueError("Supply a Docker Compose subcommand")
    args = ["docker", "compose", "-p", "deployboard-perf24"]
    for path in FILES:
        args.extend(["-f", path])
    return args + list(command)


def main():
```

```
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("--variant", choices=["baseline", "candidate"], default="baseline")
    parser.add_argument("compose_args", nargs=argparse.REMAINDER)
    options = parser.parse_args()
    if not options.compose_args:
        parser.error("Supply Compose arguments, such as config or up -d --build")
    environ = os.environ.copy()
    environ["DEPLOYBOARD_IMAGE"] = f"deployboard:ch24-{options.variant}"
    environ["DEPLOYBOARD_DOCKERFILE"] = (
        "Dockerfile.optimized" if options.variant == "candidate" else "Dockerfile"
    )
    environ["DEPLOYBOARD_HTTP_PORT"] = "18024"
    command = make_command(options.variant, options.compose_args)
    raise SystemExit(subprocess.call(command, env=environ))


if __name__ == "__main__":
    main()
```

This wrapper locks the project name and host port, then selects the build variant. Keep it in the project root and run commands from that directory; do not rename the project mid-lab. The existing Chapter 19 secret files are required, but no credential value is copied into an image or written to benchmark evidence.

Create identical production fixtures once

SOURCE FILE / tools/seed24.py

```
"""Seed exactly 20 stable production rows on the isolated localhost test stack."""
from pathlib import Path
from urllib.parse import urlsplit
from tools.bench24 import request

BASE = "http://127.0.0.1:18024"
PREFIX = "Ch24 stable record "


def main():
    assert urlsplit(BASE).hostname == "127.0.0.1"
    viewer = Path("private/viewer_token.txt").read_text().strip()
    operator = Path("private/operator_token.txt").read_text().strip()
    status, rows, _ = request(
        BASE, viewer, "GET", "/v1/deployments?environment=production")
    if status != 200:
        raise SystemExit("Production listing unavailable; check migrations and readiness")
    existing = {row["name"] for row in rows if row["name"].startswith(PREFIX)}
    for i in range(1, 21):
        name = PREFIX + f"{i:02d}"
```

```
        if name in existing:
            continue
        result, _, _ = request(BASE, operator, "POST", "/v1/deployments", {
            "name": name, "environment": "production", "version": "24.0.0"})
        if result != 201:
            raise SystemExit(f"Seeding failed for item {i}: HTTP {result}")
    status, rows, _ = request(
        BASE, viewer, "GET", "/v1/deployments?environment=production")
    created = [row for row in rows if row["name"].startswith(PREFIX)]
    if status != 200 or len(created) != 20:
        raise SystemExit("Fixture count mismatch; check database before benchmarking")
    print("PASS: 20 stable production records, no credentials printed")


if __name__ == "__main__":
    main()
```

The fixture tool recognizes twenty fixed names and can be rerun without adding duplicates. It refuses to proceed if the final row count is wrong. To protect an established DeployBoard installation, it only talks to the isolated localhost port. The host credentials stay in private/ and remain excluded from the build context.

Run bounded measurements using the complete tool

SOURCE FILE / tools/bench24.py

```
"""Bounded, authenticated localhost benchmark. Saves aggregate data only."""
import argparse
import json
import math
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
from pathlib import Path
from urllib.error import HTTPError, URLError
from urllib.parse import urlsplit
from urllib.request import Request, urlopen
from uuid import uuid4


def percentile(values, fraction):
    if not values:
        raise ValueError("No samples")
    series = sorted(values)
    return series[max(0, math.ceil(len(series) * fraction) - 1)]


def request(base, token, method, path, payload=None, timeout=8):
    data = json.dumps(payload).encode() if payload is not None else None
```

```
    headers = {"Authorization": "Bearer " + token, "Content-Type": "application/json"}
    req = Request(base + path, method=method, data=data, headers=headers)
    start = time.perf_counter()
    try:
        with urlopen(req, timeout=timeout) as response:
            body = response.read()
            parsed = json.loads(body) if body else None
            return response.status, parsed, 1000 * (time.perf_counter() - start)
    except HTTPError as error:
        return error.code, None, 1000 * (time.perf_counter() - start)
    except (TimeoutError, URLError, OSError):
        return 0, None, 1000 * (time.perf_counter() - start)


def api_benchmark(base, token, count, concurrency):
    for _ in range(5):
        status, _, _ = request(
            base, token, "GET", "/v1/deployments?environment=production")
        if status != 200:
            raise RuntimeError(f"Warm-up failed: HTTP {status}")
    started = time.perf_counter()
    samples, errors = [], 0
    with ThreadPoolExecutor(max_workers=concurrency) as pool:
        futures = [pool.submit(
            request, base, token, "GET", "/v1/deployments?environment=production")
            for _ in range(count)]
        for future in as_completed(futures):
            status, _, latency = future.result()
            samples.append(latency)
            errors += status != 200
    elapsed = time.perf_counter() - started
    return {"kind": "api", "count": count, "concurrency": concurrency,
            "errors": errors, "elapsed_s": round(elapsed, 3),
            "p50_ms": round(percentile(samples, .50), 2),
            "p95_ms": round(percentile(samples, .95), 2),
            "rps": round(count / elapsed, 2)}


def jobs_benchmark(base, viewer, operator, count, deadline):
    status, created, _ = request(base, operator, "POST", "/v1/deployments", {
        "name": "Perf lab " + uuid4().hex[:8],
        "environment": "staging", "version": "24.0.0"})
    if status != 201:
        raise RuntimeError(f"Could not create test deployment: HTTP {status}")
    ids = []
    started = time.perf_counter()
    for _ in range(count):
        key = "perf24-" + uuid4().hex
```

```
        status, job, _ = request(base, operator, "POST",
            f"/v1/deployments/{created['id']}/checks?key={key}")
        if status != 202 or not job or "id" not in job:
            raise RuntimeError(f"Job enqueue failed: HTTP {status}")
        ids.append(job["id"])
    pending = set(ids)
    failed = 0
    while pending and (time.perf_counter() - started) < deadline:
        for job_id in list(pending):
            status, job, _ = request(base, viewer, "GET", f"/v1/checks/{job_id}")
            if status != 200 or not job:
                raise RuntimeError("Job lookup failed")
            if job["status"] in ("succeeded", "failed"):
                pending.remove(job_id)
                failed += job["status"] == "failed"
        if pending:
            time.sleep(1)
    elapsed = time.perf_counter() - started
    succeeded = count - failed - len(pending)
    return {"kind": "jobs", "count": count, "concurrency": 1,
            "errors": failed + len(pending), "succeeded": succeeded,
            "unfinished": len(pending), "elapsed_s": round(elapsed, 3),
            "jobs_per_s": round(succeeded / elapsed, 3)}


def main():
    p = argparse.ArgumentParser(description=__doc__)
    p.add_argument("--mode", choices=["api", "jobs"], required=True)
    p.add_argument("--base", default="http://127.0.0.1:18024")
    p.add_argument("--count", type=int, default=40)
    p.add_argument("--concurrency", type=int, default=4)
    p.add_argument("--deadline", type=int, default=120)
    p.add_argument("--label", choices=["baseline", "candidate"], required=True)
    a = p.parse_args()
    host = urlsplit(a.base)
    if host.scheme != "http" or host.hostname not in ("127.0.0.1", "localhost"):
        p.error("Localhost HTTP only; never benchmark a public endpoint")
    if not (1 <= a.count <= 200 and 1 <= a.concurrency <= 8
            and 10 <= a.deadline <= 300):
        p.error("Use count 1-200, concurrency 1-8, deadline 10-300")
    viewer = Path("private/viewer_token.txt").read_text(encoding="utf-8").strip()
    operator = Path("private/operator_token.txt").read_text(encoding="utf-8").strip()
    if not viewer or not operator:
        p.error("Prepare both local token files first")
    base = a.base.rstrip("/")
    if a.mode == "api":
        result = api_benchmark(base, viewer, a.count, a.concurrency)
    else:
```

```
        result = jobs_benchmark(base, viewer, operator, a.count, a.deadline)
    result.update({"label": a.label, "endpoint": "localhost", "unit": "ms"})
    directory = Path("evidence/ch24")
    directory.mkdir(parents=True, exist_ok=True)
    path = directory / f"{a.label}-{a.mode}.json"
    path.write_text(json.dumps(result, indent=2) + "\n", encoding="utf-8")
    print(json.dumps(result, indent=2))
    if result["errors"]:
        raise SystemExit(f"FAIL: {result['errors']} errors; report saved to {path}")
    print("PASS: evidence saved to", path)


if __name__ == "__main__":
    main()
```

For API measurements the test path is GET /v1/deployments?environment=production; the response body is read but not printed. For jobs, the tool creates one staging deployment, submits idempotency-keyed checks, polls for completion, and counts failed or unfinished tasks as errors. This reports end-to-end completion throughput, not Redis command throughput. The JSON output contains aggregate metrics only, not bearer tokens.

24.7 Recognize Database and Queue Bottlenecks

A rising HTTP p95 with a quiet worker suggests API or database contention. A flat API p95 with queued jobs accumulating suggests publisher, Redis, or worker pressure. Those are hypotheses, not diagnoses. Correlate service logs, container resource signals, job status, and database activity before changing pool sizes or adding more workers.

Observed evidence	Likely investigation	First safe test
High API p95; database CPU busy	Slow SELECT, missing index, pool wait	Compare explain plans on a disposable fixture
API fast; queued jobs accumulate	Publisher cannot deliver or worker capacity low	Inspect publisher/worker logs and job status
Worker restarts, OOMKilled=true	Memory ceiling too low or leak	Review memory over time; reduce concurrency
High CPU with unchanged throughput	CPU throttling, lock contention	Hold workload constant; increase budget one step
Redis unavailable, records persist	Dependency outage, not data loss	Recover Redis service; verify PostgreSQL records

Avoid increasing the SQLAlchemy connection pool blindly. Each API replica and worker may create additional PostgreSQL connections, and more sessions can increase contention. A task is not complete because a Redis event was published; DeployBoard records completion in PostgreSQL.

24.8 Reduce Unnecessary Container Restarts

Restart loops hide real faults and pollute latency measurements with cold starts. Compose restart policies can restart a failed process; they cannot correct an invalid password, exhausted disk, or unreachable service. Health checks should probe actual readiness, while a dependency outage requires bounded retries and incident handling, not infinite crash-and-restart cycles.

Use Chapter 22's existing evidence commands

```
python tools/dc24.py --variant baseline ps
python tools/dc24.py --variant baseline logs --tail 80 api worker publisher
python tools/dc24.py --variant baseline events --json
# Run events only while gathering an incident; stop it with Ctrl+C.
```

Record each container's restart count and OOM flag before and after the benchmark. If restarts occur during an otherwise successful run, reject the "improvement" and fix the fault. On a dedicated benchmark stack, keep health-check intervals, graceful shutdown settings, and replicas unchanged between comparisons.

24.9 Estimate Storage, Bandwidth, and Registry Costs

Separate host-side disk use, registry storage, and data transfer. A layer may be reused by many images, so the sum of listed image sizes is not a bill. Registry upload/download bytes depend on compressed layers, retention policy, cache artifacts, and your provider. Use the calculator with your provider's published prices instead of inventing a universal Docker-hosting rate.

SOURCE FILE / tools/cost24.py

```
"""Illustrative cost calculator; enter your provider's actual USD rates."""
import argparse


def estimate(image_gib, pushes, pulls, transfer_usd_gib,
```

```
            stored_gib, storage_usd_gib_month):
    if min(image_gib, pushes, pulls, transfer_usd_gib,
           stored_gib, storage_usd_gib_month) < 0:
        raise ValueError("Inputs cannot be negative")
    transfer_gib = image_gib * (pushes + pulls)
    return {"transfer_gib": round(transfer_gib, 3),
            "transfer_usd": round(transfer_gib * transfer_usd_gib, 2),
            "storage_usd_month": round(stored_gib * storage_usd_gib_month, 2)}


def main():
    p = argparse.ArgumentParser(description=__doc__)
    for key in ("image-gib", "pushes", "pulls", "transfer-usd-gib",
                "stored-gib", "storage-usd-gib-month"):
        p.add_argument("--" + key, type=float, required=True)
    args = vars(p.parse_args())
    result = estimate(*(args[k.replace("-", "_")] for k in (
        "image-gib", "pushes", "pulls", "transfer-usd-gib",
        "stored-gib", "storage-usd-gib-month")))
    for key, value in result.items():
        print(f"{key}: {value}")


if __name__ == "__main__":
    main()
```

Illustrative input only; replace all prices with your provider's rates

```
python tools/cost24.py --image-gib 0.2 --pushes 2   --pulls 10 --transfer-usd-gib 0.09   --stored-gib
0.5 --storage-usd-gib-month 0.02
```

The multiline example uses Bash backslashes. In PowerShell, enter those arguments on one line without backslashes. The same rule applies to the report comparison commands below.

The example estimates 2.4 GiB of total transfer (0.2 × 12), $0.22 after rounding, and $0.01/month for 0.5 GiB stored. These are intentionally hypothetical rates, not a quotation from a registry provider. The estimate excludes CI compute minutes, host energy, regional egress rules, free allowances, retention, and layer deduplication.

24.10 Prove Improvements with Repeatable Tests

Run the functional tests before and after changing the build. Performance acceptance requires no errors, no new restarts, and no material regression. For this lab, the exploratory gate permits no more than 5% degradation in

API p95 and no more than 5% reduction in completed jobs/s. Passing the gate means “not measurably worse by this tolerance,” not proof of a statistically significant speed-up.

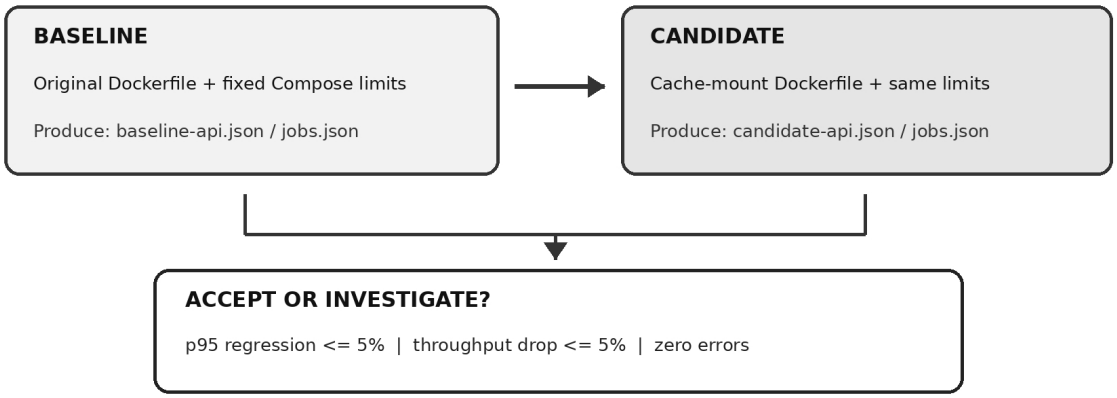


Figure 24.2 | Identical workloads feed a conservative comparison gate.

SOURCE FILE / tools/report24.py

```
"""Compare two compatible benchmark files without inventing results."""
import argparse
import json
from pathlib import Path


def assess(before, after):
    if before["kind"] != after["kind"] or before["count"] != after["count"]:
        raise ValueError("Benchmark kind and request count must match")
    if before["concurrency"] != after["concurrency"]:
        raise ValueError("Concurrency must match")
    metric = "p95_ms" if before["kind"] == "api" else "jobs_per_s"
    baseline, candidate = float(before[metric]), float(after[metric])
    if baseline <= 0:
        raise ValueError("Baseline measure must be positive")
    change = 100 * (candidate - baseline) / baseline
    ok = before["errors"] == 0 and after["errors"] == 0
    if metric == "p95_ms":
        ok = ok and candidate <= baseline * 1.05
    else:
        ok = ok and candidate >= baseline * .95
    return {"kind": before["kind"], "metric": metric,
            "baseline": baseline, "candidate": candidate,
            "change_percent": round(change, 2), "gate": "PASS" if ok else "REVIEW"}


def main():
    p = argparse.ArgumentParser(description=__doc__)
```

```
    p.add_argument("baseline", type=Path)
    p.add_argument("candidate", type=Path)
    a = p.parse_args()
    before = json.loads(a.baseline.read_text(encoding="utf-8"))
    after = json.loads(a.candidate.read_text(encoding="utf-8"))
    result = assess(before, after)
    print(json.dumps(result, indent=2))
    if result["gate"] != "PASS":
        raise SystemExit(1)


if __name__ == "__main__":
    main()
```

The compare tool rejects mismatched counts and concurrency, reports a signed percent change, and exits nonzero when the candidate breaches its tolerance or has errors. It does not manufacture benchmark output; the JSON files appear only after successful local runs.

Local tests for the new tooling

SOURCE FILE / tests/test_perf24.py

```
"""Local deterministic checks; full runtime tests require Docker Engine."""
import pytest
from tools.bench24 import percentile
from tools.cost24 import estimate
from tools.dc24 import make_command
from tools.report24 import assess


def test_nearest_rank_percentiles():
    assert percentile([20, 10, 30, 40], .50) == 20
    assert percentile([20, 10, 30, 40], .95) == 40


def test_report_rejects_mismatched_workloads():
    before = {"kind": "api", "count": 10, "concurrency": 2, "p95_ms": 100, "errors": 0}
    with pytest.raises(ValueError):
        assess(before, {**before, "concurrency": 5})


def test_report_fails_regressions():
    before = {"kind": "api", "count": 10, "concurrency": 2, "p95_ms": 100, "errors": 0}
    assert assess(before, {**before, "p95_ms": 110})["gate"] == "REVIEW"
    assert assess(before, {**before, "p95_ms": 98})["gate"] == "PASS"
    assert assess(before, {**before, "errors": 1})["gate"] == "REVIEW"
    assert assess({**before, "errors": 1}, before)["gate"] == "REVIEW"
```

```
def test_cost_estimate_and_refuse_negative():
    assert estimate(.2, 1, 4, .1, .5, .02)["transfer_usd"] == .1
    with pytest.raises(ValueError):
        estimate(-1, 1, 1, 1, 1, 1)


def test_wrapper_targets_isolated_compose_project():
    command = make_command("baseline", ["config", "--quiet"])
    assert command[:4] == ["docker", "compose", "-p", "deployboard-perf24"]
    assert command[-2:] == ["config", "--quiet"]
```

Run the project's existing Chapter 21 quality gate as well; the new unit tests validate calculation and isolation logic, not container behavior. Do not call a release approved until both the runtime acceptance and security checks pass. If candidate throughput is better but failures increase, reject it.

24.11 Checkpoint: Publish a Measured Optimization Report

This final lab produces a traceable, reproducible claim. Keep the project isolated, run the migrations before the first benchmark, warm the services, and measure API and worker paths. The commands below are intended for a terminal opened in the cumulative source directory and require an active Docker Engine.

Sequence A — Baseline

Prepare secret files only if the three credential files are missing. Unlike a naive all-at-once startup, this order runs migrations before asking the proxy to depend on API readiness, which checks the actual tables.

```
python tools/prepare_ch19_secrets.py  # only if credential files are absent
python tools/dc24.py --variant baseline config --quiet
python tools/dc24.py --variant baseline build api
python tools/dc24.py --variant baseline up -d db redis
python tools/dc24.py --variant baseline run --rm migrate
python tools/dc24.py --variant baseline up -d api publisher worker proxy
python -m tools.seed24
python tools/bench24.py --mode api --label baseline
python tools/bench24.py --mode jobs --label baseline --count 10
```

Sequence B — Candidate, same volume and workload

```
python tools/dc24.py --variant candidate up -d --build
python tools/dc24.py --variant candidate ps
python tools/bench24.py --mode api --label candidate
```

```
python tools/bench24.py --mode jobs --label candidate --count 10
```

Sequence C — Compare, accept, or investigate

```
python tools/report24.py evidence/ch24/baseline-api.json   evidence/ch24/candidate-api.json
python tools/report24.py evidence/ch24/baseline-jobs.json   evidence/ch24/candidate-jobs.json
# Preserve volumes; no -v and no system-wide prune.
python tools/dc24.py --variant candidate down
```

If preparation scripts find existing secret files, do not overwrite them. If the candidate build fails, return to the baseline variant using the same wrapper and preserve the isolated database volume for diagnosis. Rerun the quality gate after changing any base image or dependency; record a new Chapter 20 SBOM and vulnerability review for a new release candidate.

Optimization report — fill from actual evidence

Field	What to record
Identity	Date, operator, Docker/Buildx/Compose versions, architecture
Change	Baseline Dockerfile → Dockerfile.optimized pip-cache mount
Build evidence	First/repeat elapsed seconds; original/candidate image bytes
Runtime evidence	Baseline/candidate API p50/p95, RPS, error count
Async evidence	Completed checks/s, failures, pending jobs, retries
Operating envelope	Limits, restart counts, observed CPU/memory, dataset size
Cost	Provider rates, estimated transfer, stored GiB-month, exclusions
Gates	Chapter 21 tests; Chapter 20 review; Chapter 24 regression gate
Decision	APPROVE / REJECT / INVESTIGATE + date, evidence filenames

Acceptance proof

- Twenty stable production fixture records exist and the auth-protected API responds through the proxy.
- Both variants use one isolated Compose project, the same data set, and equal CPU/memory limits.
- Both modes finish with zero errors; measured evidence files are saved for each variant.
- The comparison scripts return PASS (or the candidate is explicitly rejected with evidence).
- No new restart loop or OOM kill occurs, and the earlier security and functional gates remain green.

Your optimization report is the outcome of this chapter. Retain the unchanged raw evidence and justify any claimed speed or cost benefit with measurements. The next chapters can use this measured local release as the reference point for reliable CI/CD and deployment operations.

PART VII

PUBLISH, AUTOMATE, AND DEPLOY RELEASES

Chapter 25 — Publish and Manage Container Images in Registries

A release is not complete when Docker builds an image on your laptop. You must be able to locate it, control who can fetch it, prove which bytes you shipped, and retrieve those same bytes on another host. This chapter turns DeployBoard's tested application image into a traceable registry artifact—not a promise attached to a mutable tag.

The release result

You will publish a release-candidate image to a repository you control; record the top-level registry digest; verify the tag has not moved; pull that immutable reference elsewhere; and preserve a human-review release sheet. No GitHub repository is required. An OCI-compatible registry account and Docker Engine are required for the live publication steps.

SCOPE AND SAFETY Your starting point is the complete DeployBoard application after Chapter 24. Keep PostgreSQL, Redis, Caddy, local secret files, and application state on their existing Compose projects. You are publishing only the API application image (also reused by its publisher, worker, and migration services)—not copying a live database into a registry.

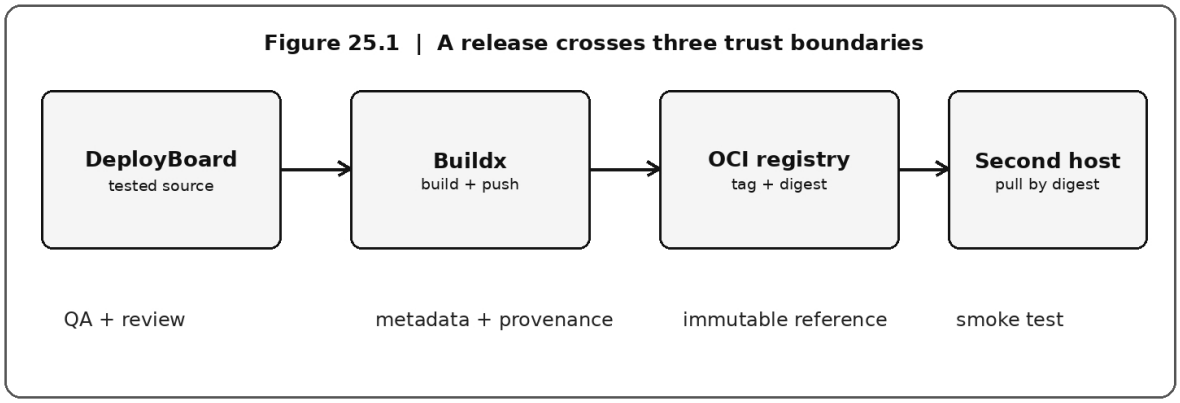


Figure 25.1 | The release path: tested source → build and push → registry → exact-digest pull.

25.1 Choose Between Docker Hub and Other OCI Registries

An image registry stores OCI images and descriptors; an image repository organizes related versions. Docker Hub is the simplest primary path

because Docker CLI login, tags, and image pages fit the beginner workflow. GitHub Container Registry (GHCR), AWS ECR, Azure Container Registry, and Google Artifact Registry provide other authentication, ownership, retention, pricing, and policy models. The CLI pattern is similar; identity and access controls are not interchangeable.

Registry	Good fit	Decision to confirm
Docker Hub	Independent distribution and a straightforward first publication	Namespace, private access, rate/plan limits and tag policy
GHCR (ghcr.io)	Image packages alongside a GitHub organization	Package visibility, permissions and workflow token access
Cloud-managed OCI registry	Deployment near a chosen cloud platform	IAM role scoping, region, lifecycle rules and transfer costs

For the core lab, create a repository named deployboard-api under your own Docker Hub namespace. Choose private unless the source and its dependencies are intentionally distributable. A private repository still requires least-privilege permissions; “private” is not an integrity guarantee.

25.2 Understand Registry Authentication and Access Tokens

An interactive login stores Docker client credentials through the configured credential helper, or less securely in a Docker configuration file if no helper is installed. Prefer a short-lived, narrowly scoped access token rather than reusing a high-privilege account password. Never paste tokens into a Dockerfile, Compose file, CLI argument, screenshot, build log, or example in this book.

Log in interactively without printing your token

```
docker login --username YOUR_NAMESPACE
# Enter a token only when prompted; never enter a real token in this book.
docker info
# docker logout   # Use after the session if appropriate.
```

On GHCR, personal access tokens (classic) have package scopes such as read:packages and write:packages; in GitHub Actions, the repository-scoped GITHUB_TOKEN is generally preferred when permitted. On Docker Hub, token permission capabilities depend on account and

organization settings. Distinguish three identities: one publisher with write permission, a release reviewer with read/metadata access, and a deployment host with read-only pull access.

CREDENTIAL BOUNDARY For automated jobs, pass a credential securely to docker login --password-stdin using your CI secret facility. Do not store it in the repository or print the environment. If your machine lacks a credential helper, inspect ~/.docker/config.json and configure secure storage before handling production tokens.

25.3 Establish a Clear Release Tagging Convention

A version tag is a human label, not immutable proof. Use an explicit convention such as vMAJOR.MINOR.PATCH-rcN for candidates and vMAJOR.MINOR.PATCH for approved releases. The example tag v1.0.0-rc1 is illustrative. Do not recycle it after a changed build; issue v1.0.0-rc2 or a new release version. Never silently repoint an image already approved and deployed.

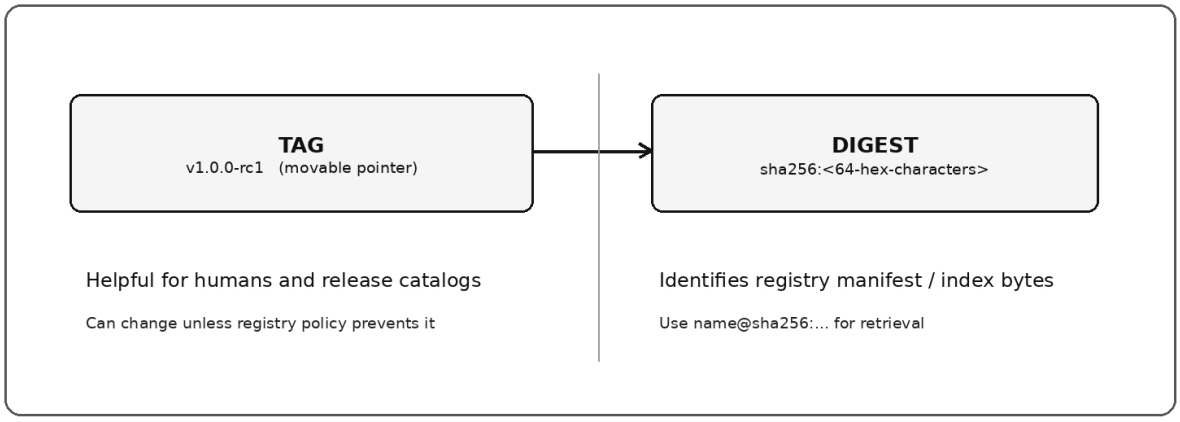


Figure 25.2 | Tags identify releases for people; registry digests identify manifest content.

Identifier	Example	Use
Candidate tag	v1.0.0-rc1	Test publication and review; never imply approval
Approved version	v1.0.0	Human release catalog after a separate approval decision
Digest reference	repo@sha256:<actual-64-hex>	Pin the exact registry manifest or index for distribution

The local Docker image ID refers to the image configuration; the registry digest refers to a manifest or multi-platform index. They are different

objects and must not be compared as if they were interchangeable. Enable tag immutability when your registry supports it, but still record and deploy by digest.

25.4 Push and Pull a Versioned DeployBoard Image

Before pushing, run the existing Chapter 21 quality gate and Chapter 20 vulnerability/SBOM review. These checks produce real evidence only when executed and recorded. Confirm the .dockerignore excludes private/, evidence/, test caches, and local credentials. For the release image, reuse Dockerfile.optimized from Chapter 24, which installs the same runtime dependencies and copies the same application files.

Build, publish, and capture Buildx metadata (run at DeployBoard root)

```
docker version
docker buildx version
# Create the evidence directory from Explorer, File Manager, or Python:
python -c "from pathlib import Path; Path('evidence/ch25').mkdir(parents=True, exist_ok=True)"
docker buildx build --platform linux/amd64 -f Dockerfile.optimized \
  --tag docker.io/YOUR_NAMESPACE/deployboard-api:v1.0.0-rc1 \
  --metadata-file evidence/ch25/build-metadata.json --push .
```

The two physical-line continuations above use a Bash-style backslash. In Windows PowerShell, enter the buildx build command on one line (without backslashes). Replace YOUR_NAMESPACE with your actual Docker Hub namespace. This contacts a registry and may incur storage or transfer charges; it is not an offline demonstration. If the registry repository does not exist or your token lacks write permission, the push must fail—fix access instead of bypassing review.

Confirm the newly published reference

```
docker buildx imagetools inspect \
  docker.io/YOUR_NAMESPACE/deployboard-api:v1.0.0-rc1
# Look for a top-level Digest: sha256:... line.
```

A push can attach provenance metadata, so the registry may hold an OCI index even for a single requested platform. This is normal. Record the top-level digest and inspect platform descriptors rather than assuming every digest in the output means the same object.

25.5 Inspect a Pushed Image by Digest

The Buildx metadata file records containerimage.digest. Compare that value with the top-level Digest returned by docker buildx imagetools inspect for the pushed tag. A mismatch can indicate a changed tag, wrong repository, registry transformation, or an incorrect evidence file. Stop the release until explained; do not “repair” the record by overwriting evidence.

Create an immutable release record using the included utility

```
python tools/registry25.py record \
  --image docker.io/YOUR_NAMESPACE/deployboard-api:v1.0.0-rc1
python tools/registry25.py verify --pull --smoke
```

The record command refuses to overwrite an existing release.json and exits nonzero if the built and remote digests differ. The verify command checks the remote tag against that saved digest, then optionally pulls name@sha256:digest and runs an isolated dependency-import smoke test with --network none. It does not certify all API behavior or authorize a production rollout.

EVIDENCE INTEGRITY Never publish a sample digest as if it were measured. The release record begins at PENDING_MANUAL_APPROVAL until a person reviews the test, scan, SBOM, and access-control evidence.

25.6 Use Private Repositories and Minimum Required Permissions

For a private repository, configure a writer identity only on a controlled publishing workstation or CI job and a separate read-only identity on the retrieval host. Confirm the recipient can pull the exact image but cannot overwrite tags, change visibility, or delete historical release artifacts. Account-level privileges, package roles, and organization membership differ among registries; verify the actual repository access screen rather than assuming the token scope alone is sufficient.

- Publisher: write permission to exactly one image repository; separate expiration and rotation plan.
- Consumer: read-only permission to pull by digest; no general registry administration.
- Review workflow: recorded evidence can be read without exposing credentials or private configuration.
- If the image must become public later, obtain approval before changing its visibility; private base layers are not automatically scrubbed.

Treat unauthorized pulls as failures, not as prompts to turn the repository public. Document who is allowed to retrieve a release and when that access expires.

25.7 Handle Multi-Platform Manifests and Registry Support

One registry tag can resolve to an image index containing different manifests for linux/amd64 and linux/arm64. The index digest identifies the set; each architecture has its own child digest. A pull chooses a compatible platform automatically if one exists, but compatibility must be tested: native modules, architecture-specific packages, and resource behavior may differ.

Optional advanced build: inspect first, then publish if both platforms are supported

```
docker buildx ls
docker buildx build --platform linux/amd64,linux/arm64 \
  -f Dockerfile.optimized \
  --tag docker.io/YOUR_NAMESPACE/deployboard-api:v1.0.0-rc2 \
  --metadata-file evidence/ch25/multi-build-metadata.json --push .
docker buildx imagetools inspect \
  docker.io/YOUR_NAMESPACE/deployboard-api:v1.0.0-rc2
```

Use a NEW tag because the multi-platform index digest will not equal the earlier single-platform reference. QEMU emulation can be significantly slower than native builders and is not automatically available. Confirm registry support for OCI indexes and attestations, then independently test the runtime import and API functions on each intended architecture. The main release record remains for rc1 unless you capture an additional evidence file and issue a new review.

25.8 Plan Image Retention and Cleanup

Image retention is a recovery policy as well as a cost policy. Retain the currently deployed digest, the last known-good rollback digest, every digest under an active incident or audit hold, and any image referenced by an open release decision. Only after confirming those dependencies should you consider deleting obsolete candidate tags or registry objects. A registry may keep untagged blobs until asynchronous garbage collection, and deleting a tag can make a retained digest unavailable depending on provider behavior.

Retention class	Keep until	Never automate before

Retention class	Keep until	Never automate before
Active production digest	Replaced and rollback window closes	Running deployments and approval record reviewed
Last known good	New candidate proven and incident-free	Rollback plan identifies a different tested digest
Release candidate	Decision, audit, and investigation completed	Vulnerability evidence and sign-off archived
Test-only temporary tag	Lab cleanup authorized	Check that no user or system depends on the tag

For local disk cleanup, distinguish docker image prune (host cache) from remote registry lifecycle rules. Do not run docker compose down -v or docker volume prune: deleting live PostgreSQL or Redis volumes has no relationship to cleaning published image tags.

25.9 Avoid Mutable Latest Tags in Production Releases

The tag latest is a convenience default when no tag is supplied, not a guarantee of recency, safety, or approval. Do not publish DeployBoard production deployments as image: latest. Use an explicit version tag for human review and the recorded image@sha256:... reference for the actual deployment. Where possible, enable immutable version tags on Docker Hub or equivalent policy in your selected registry.

A reliable pull and a misleading pull

```
# Reliable: replace with a REAL digest captured from your registry.
docker pull docker.io/YOUR_NAMESPACE/deployboard-api@sha256:<REAL_DIGEST>
# Unsafe for production reproducibility:
# docker pull docker.io/YOUR_NAMESPACE/deployboard-api:latest
```

A pinned digest intentionally does not update itself after security fixes. When the base image or dependencies change, rebuild, rescan, retest, publish under a new version, and record the new digest. Reproducibility and patching require an explicit release process rather than contradictory auto-updates.

25.10 Document the Image Publication Process

A repeatable release needs a short chain of custody: source reviewed, test gate passed, build input controlled, image pushed, tag resolved to digest,

vulnerability/SBOM evidence reviewed, access checked, and digest verified on a separate host. The following Python program is the complete, standalone Chapter 25 registry evidence tool, with no nonstandard Python imports. Place it at tools/registry25.py. The companion ZIP includes the same file and tests.

Complete source — tools/registry25.py

```
"""Record and verify a DeployBoard registry image without storing credentials.

Run from the DeployBoard project root. Requires Python 3.11+ and Docker Buildx.
"""
import argparse
import json
import re
import subprocess
from datetime import datetime, timezone
from pathlib import Path

DIGEST = re.compile(r"^sha256:[0-9a-f]{64}$")
TAG = re.compile(r"^[a-z0-9][a-z0-9._/-]+:[a-zA-Z0-9_][a-zA-Z0-9_.-]*$")


def checked_digest(value):
    if not isinstance(value, str) or not DIGEST.fullmatch(value):
        raise ValueError("Expected a complete sha256 registry digest")
    return value


def checked_tag(value):
    if not TAG.fullmatch(value) or value.endswith(":latest"):
        raise ValueError("Use a fully qualified, versioned image tag; not :latest")
    if "/" not in value.rsplit(":", 1)[0]:
        raise ValueError("Include registry namespace and repository")
    return value


def manifest_digest(output):
    for line in output.splitlines():
        match = re.match(r"^Digest:\s*(sha256:[0-9a-f]{64})\s*$", line)
        if match:
            return checked_digest(match.group(1))
    raise ValueError("No top-level registry manifest digest was returned")


def docker(*args):
    result = subprocess.run(["docker", *args], text=True, capture_output=True)
    if result.returncode:
```

```
        raise RuntimeError("Docker registry command failed (exit "
                           f"{result.returncode}); check login, network and permissions")
    return result.stdout


def registry_digest(tag):
    return manifest_digest(docker("buildx", "imagetools", "inspect", tag))


def capture(tag, metadata_path, out_path):
    tag = checked_tag(tag)
    data = json.loads(Path(metadata_path).read_text(encoding="utf-8"))
    built = checked_digest(data.get("containerimage.digest"))
    remote = registry_digest(tag)
    if built != remote:
        raise ValueError("Build metadata and remote tag digests differ: STOP")
    record = {
        "schema": "deployboard.registry-release.v1",
        "tag_ref": tag,
        "registry_digest": remote,
        "immutable_ref": tag.rsplit(":", 1)[0] + "@" + remote,
        "recorded_at_utc": datetime.now(timezone.utc).isoformat(),
        "review_status": "PENDING_MANUAL_APPROVAL",
    }
    dest = Path(out_path)
    dest.parent.mkdir(parents=True, exist_ok=True)
    if dest.exists():
        raise FileExistsError("Release record already exists; do not overwrite")
    dest.write_text(json.dumps(record, indent=2) + "\n", encoding="utf-8")
    print("Recorded immutable reference:", record["immutable_ref"])


def verify(record_path, pull=False, smoke=False):
    record = json.loads(Path(record_path).read_text(encoding="utf-8"))
    tag = checked_tag(record["tag_ref"])
    expected = checked_digest(record["registry_digest"])
    ref = tag.rsplit(":", 1)[0] + "@" + expected
    if record["immutable_ref"] != ref:
        raise ValueError("Release record is inconsistent")
    if registry_digest(tag) != expected:
        raise ValueError("Tag has moved or release record is wrong: STOP")
    print("PASS: registry tag resolves to recorded digest")
    if pull or smoke:
        docker("pull", ref)
        print("PASS: exact digest was pulled")
    if smoke:
        docker("run", "--rm", "--network", "none", "--entrypoint", "python",
               ref, "-c", "import fastapi, sqlalchemy; print('runtime imports OK')")
```

```
        print("PASS: isolated Python dependency smoke test")


def main():
    p = argparse.ArgumentParser(description=__doc__)
    subs = p.add_subparsers(dest="action", required=True)
    rec = subs.add_parser("record")
    rec.add_argument("--image", required=True)
    rec.add_argument("--metadata", default="evidence/ch25/build-metadata.json")
    rec.add_argument("--out", default="evidence/ch25/release.json")
    check = subs.add_parser("verify")
    check.add_argument("--record", default="evidence/ch25/release.json")
    check.add_argument("--pull", action="store_true")
    check.add_argument("--smoke", action="store_true")
    args = p.parse_args()
    try:
        if args.action == "record":
            capture(args.image, args.metadata, args.out)
        else:
            verify(args.record, args.pull, args.smoke)
    except (OSError, ValueError, RuntimeError, KeyError, json.JSONDecodeError) as exc:
        p.exit(1, f"FAIL: {exc}\n")


if __name__ == "__main__":
    main()
```

The utility deliberately never requests or echoes a registry token. Authentication is performed separately by the Docker CLI. It rejects non-versioned and latest tags, fails if the registry returns a different top-level digest, and creates a fresh—not silently rewritten—release record. Attach the human-approval checklist in release/IMAGE_RELEASE_CHECKLIST.txt and link the Chapter 20 scan and SBOM evidence before approving the actual release.

25.11 Checkpoint: Retrieve an Identical Release Elsewhere

Use a second Docker host with the same required CPU architecture and a read-only registry identity. Transfer tools/registry25.py and your REAL evidence/ch25/release.json—not the source tree, database volume, build secrets, or image archive. From the directory containing the transferred

evidence, run the exact-digest verification. If your registry is private, perform docker login with pull-only credentials on that second host first.

Second-host acceptance proof

```
python tools/registry25.py verify \
  --record evidence/ch25/release.json --pull --smoke
# Expected successful messages:
# PASS: registry tag resolves to recorded digest
# PASS: exact digest was pulled
# PASS: isolated Python dependency smoke test
```

On Windows PowerShell, combine continuation lines into a single command. Success proves that the saved tag and registry digest still agree and the exact image imports its key Python packages without network access. It does not prove the entire DeployBoard stack is healthy: Chapter 21's API/database/worker integration gate must still pass in the deployment environment before release approval.

Tests for the new evidence utility — tests/test_registry25.py

```
"""Pure unit tests: no registry credentials or Docker daemon required."""
import json
from pathlib import Path
from unittest.mock import patch

import pytest
from tools import registry25

D1 = "sha256:" + "a" * 64
D2 = "sha256:" + "b" * 64
TAG = "docker.io/example/deployboard-api:v1.0.0-rc1"


def test_digest_and_tag_validation():
    assert registry25.checked_digest(D1) == D1
    assert registry25.checked_tag(TAG) == TAG
    with pytest.raises(ValueError):
        registry25.checked_digest("sha256:123")
    with pytest.raises(ValueError):
        registry25.checked_tag("deployboard:latest")


def test_top_level_manifest_digest_only():
    text = "Name: example\nMediaType: application/vnd.oci.image.index.v1+json\nDigest: " + D1
    assert registry25.manifest_digest(text) == D1
    with pytest.raises(ValueError):
        registry25.manifest_digest("Platforms: linux/amd64\nDigest: incorrect")
```

```
def test_capture_creates_immutable_record(tmp_path):
    meta = tmp_path / "build.json"
    meta.write_text(json.dumps({"containerimage.digest": D1}))
    output = tmp_path / "release.json"
    with patch.object(registry25, "registry_digest", return_value=D1):
        registry25.capture(TAG, meta, output)
    record = json.loads(output.read_text())
    assert record["immutable_ref"] == "docker.io/example/deployboard-api@" + D1
    assert record["review_status"] == "PENDING_MANUAL_APPROVAL"
    with pytest.raises(FileExistsError):
        with patch.object(registry25, "registry_digest", return_value=D1):
            registry25.capture(TAG, meta, output)


def test_capture_detects_registry_mismatch(tmp_path):
    meta = tmp_path / "build.json"
    meta.write_text(json.dumps({"containerimage.digest": D1}))
    with patch.object(registry25, "registry_digest", return_value=D2):
        with pytest.raises(ValueError, match="differ"):
            registry25.capture(TAG, meta, tmp_path / "release.json")


def test_verify_refuses_mutated_tag(tmp_path):
    record = tmp_path / "release.json"
    record.write_text(json.dumps({"tag_ref": TAG, "registry_digest": D1,
        "immutable_ref": TAG.rsplit(":", 1)[0] + "@" + D1}))
    with patch.object(registry25, "registry_digest", return_value=D2):
        with pytest.raises(ValueError, match="moved"):
            registry25.verify(record)


def test_verify_pull_uses_digest_not_tag(tmp_path):
    record = tmp_path / "release.json"
    immutable = TAG.rsplit(":", 1)[0] + "@" + D1
    record.write_text(json.dumps({"tag_ref": TAG, "registry_digest": D1,
        "immutable_ref": immutable}))
    with patch.object(registry25, "registry_digest", return_value=D1):
        with patch.object(registry25, "docker", return_value="") as call:
            registry25.verify(record, pull=True)
    call.assert_called_once_with("pull", immutable)
```

Run the unit tests without registry access

```
python -m pytest -q tests/test_registry25.py
```

Expected local result: six tests pass. These mock the registry and therefore cannot certify a real push, scan, or pull. For a complete local quality gate, also run the Chapter 21 script and record its actual result. An image is

approved only after the human checklist is complete and all required production evidence exists.

FAILURE RESPONSE Do not replace a failed digest check with docker pull :latest or mutate release.json to match a changed tag. Issue a new candidate release, investigate what changed, and keep the original release record for the audit trail.

Acceptance proof — fill with measured facts, never illustrative values

Evidence item	What you must record
Publication	Repository visibility, image tag, Buildx command and UTC time
Identity	Publisher write scope and second-host read-only scope
Integrity	Build digest, remote top-level digest, and immutable image reference
Risk review	Chapter 20 scan/SBOM findings and signed exception decisions
Functional quality	Chapter 21 full gate output; Chapter 25 import smoke is supplementary
Recovery	Last known-good digest, retention hold, and rollback test status
Approval	Human reviewer, date, decision, and unresolved limitations

Release approval worksheet — leave blank until you complete the live work

Source revision or source snapshot ID: _____

Registry image tag: _____

Verified registry digest (sha256): _____

Chapter 20 scan / SBOM evidence location:

_____

Chapter 21 quality-gate result and timestamp: _____

Second-host immutable pull and smoke outcome: _____

Reviewer / decision / date: _____

You now have a method for creating a traceable image release artifact—not merely a successful local build. In Chapter 26, you will apply this evidence to a controlled, automated release pipeline.

Chapter 26 — Build a Docker CI Pipeline with GitHub Actions

DeployBoard now builds locally and can be published by digest. The remaining risk is process: someone could publish a revision that was never tested, expose a registry token to an untrusted pull request, or mistake a green build for a healthy application. In this chapter you turn the Chapter 21 quality gate and Chapter 25 release practices into one controlled CI path.

You will produce A complete GitHub Actions workflow, a local policy validator, seven focused regression tests, a repeatable no-GitHub quality gate, and a conditional GHCR release that writes an immutable digest to the build log.

Required The book-supplied DeployBoard source from Chapter 25 (included in the ZIP), Python 3.11+, Git, Docker Engine and Docker Compose for live labs. A GitHub account is optional; exercises 26.2–26.8 work locally.

Release rule Only a version-tag push may publish. A pull request, manual run, or ordinary main-branch push never receives registry-publishing authority.

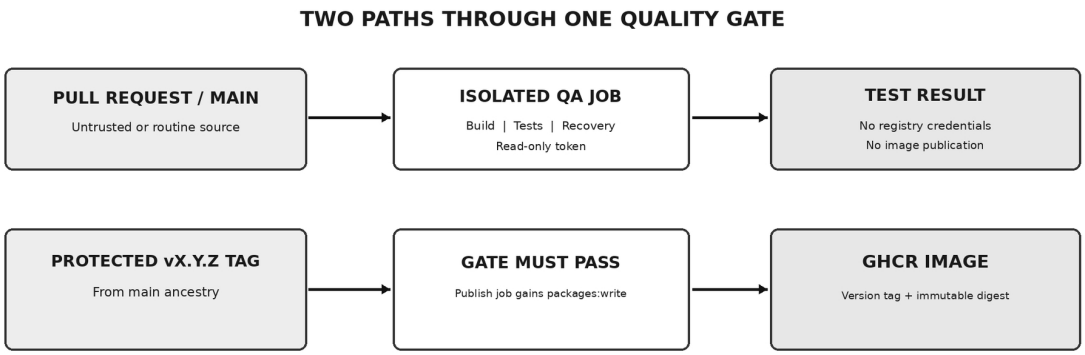


Figure 26.1 One tested commit, two different permissions and outcomes.

26.1 Understand the CI Pipeline and Its Trust Boundaries

A CI job executes repository-controlled instructions. That means reviewing a pull request is not the same trust decision as granting credentials to a release. Your design uses GitHub-hosted disposable runners, minimal token permissions, and a separate publish job that depends on a successful quality gate.

Event	What runs	Registry access
Pull request to main	Static checks, disposable Compose tests	None; no publish job
Push to main	The same quality gate	None; no publish job
Manual workflow run	Quality gate only	None; no publish job
Push of protected vX.Y.Z tag	Quality gate, then a tagged release job	Scoped GitHub token in publish only

The workflow uses pull_request rather than pull_request_target. Do not run untrusted pull-request code on a privileged self-hosted runner or grant forked pull requests registry secrets. Publishing is deliberately confined to a separate job with packages:write permission.

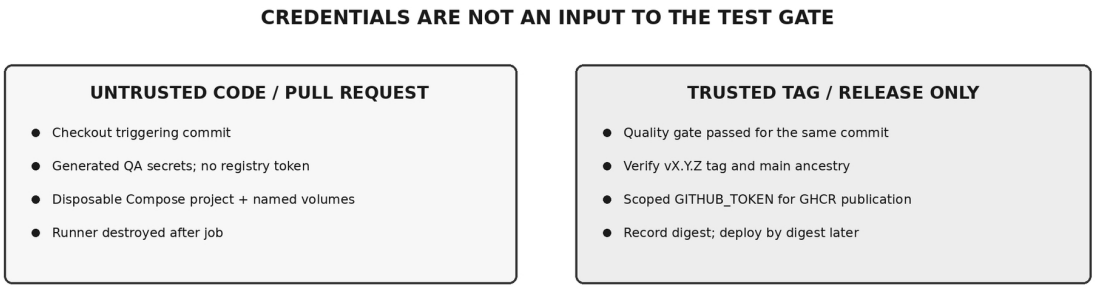


Figure 26.2 The CI trust boundary: credential-free testing before controlled publication.

26.2 Recreate the Repository from Book-Supplied Project Files

Unpack DeployBoard_Chapter_26_Source from the companion ZIP into an empty working folder. Its Python API, migrations, Redis worker, Caddy proxy, tests, Dockerfiles, and previous Compose overlays are the same application used in Chapters 14–25. The GitHub workflow is an additional file, not a replacement for that stack.

Essential project tree (the ZIP contains the entire source)

DeployBoard_Chapter_26_Source/

```
.github/workflows/deployboard-ci.yml
app/  migrations/  tests/  tools/  proxy/  site/
Dockerfile  Dockerfile.test  Dockerfile.test.dockerignore
compose.yaml  compose.harden.yaml  compose.qa.yaml
requirements.txt  requirements-dev.txt  requirements-ci.txt
.dockerignore  .gitignore  .env.example
```

Create a local repository — no GitHub connection required

```
cd DeployBoard_Chapter_26_Source
git init -b main
git add .
git status --short
git commit -m "Add DeployBoard Docker CI quality gate"
```

Before committing, check that private/, qa-private/, .env, evidence/, and generated files are absent from the staged list. The companion archive includes README placeholders for secret directories, not actual secret values. Never commit real credentials, release digests from private registries, or local caches.

Optional remote Create an empty, private GitHub repository, connect it with git remote add origin <YOUR-REPOSITORY-URL>, and git push -u origin main. No preexisting public GitHub repository is required to complete the local exercises.

26.3 Create the Workflow Configuration from Scratch

Save the following complete file as .github/workflows/deployboard-ci.yml. It uses the current major versions documented by Docker and GitHub at manuscript preparation. These major action tags are movable: for a real production repository, replace them with verified immutable commit SHAs or a workflow dependency lockfile after testing the chosen revisions.

FILE .github/workflows/deployboard-ci.yml — complete listing

```
# Chapter 26 — CI for DeployBoard. Requires a GitHub repository to execute.
# PRs have no publish job. Version-tag publishing requires a passing gate.
name: DeployBoard CI and release

on:
  pull_request:
    branches: [main]
  push:
    branches: [main]
```

```
    tags: ['v*']
  workflow_dispatch:

permissions:
  contents: read

concurrency:
  group: deployboard-${{ github.ref }}
  cancel-in-progress: false

jobs:
  gate:
    name: Test in disposable containers
    runs-on: ubuntu-24.04
    timeout-minutes: 25
    steps:
      - name: Check out exact triggering revision
        uses: actions/checkout@v6
      - name: Prepare workflow validator
        shell: bash
        run: |
          python3 -m venv .ci-venv
          .ci-venv/bin/python -m pip install -r requirements-ci.txt
      - name: Validate workflow and repository inputs
        run: .ci-venv/bin/python tools/ci26_validate.py
      - name: Run isolated Docker quality gate
        run: python3 tools/quality_gate.py --cleanup

  publish:
    name: Publish approved tag to GHCR
    needs: [gate]
    if: >-
      ${{ github.event_name == 'push' && startsWith(github.ref, 'refs/tags/v') }}
    runs-on: ubuntu-24.04
    timeout-minutes: 25
    permissions:
      contents: read
      packages: write
    steps:
      - name: Check out release revision and branch history
        uses: actions/checkout@v6
        with:
          fetch-depth: 0
      - name: Verify semantic release tag and main ancestry
        id: release
        shell: bash
        run: |
          if [[ ! "$GITHUB_REF_NAME" =~ ^v[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
            echo 'Release tags must look like v1.2.3' >&2
```

```
      exit 1
    fi
    git fetch --no-tags origin main
    if ! git merge-base --is-ancestor "$GITHUB_SHA" origin/main; then
      echo 'Release commit is not reachable from main' >&2
      exit 1
    fi
    echo "image=ghcr.io/${GITHUB_REPOSITORY,,}" >> "$GITHUB_OUTPUT"
    echo "version=${GITHUB_REF_NAME#v}" >> "$GITHUB_OUTPUT"
- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v4
- name: Authenticate to GitHub Container Registry
  uses: docker/login-action@v4
  with:
    registry: ghcr.io
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}
- name: Build and push digest-addressable image
  id: build
  uses: docker/build-push-action@v7
  with:
    context: .
    file: ./Dockerfile
    platforms: linux/amd64
    push: true
    tags: |
      ${{ steps.release.outputs.image }}:${{ steps.release.outputs.version }}
      ${{ steps.release.outputs.image }}:sha-${{ github.sha }}
    cache-from: type=gha,scope=deployboard-api-release
    cache-to: type=gha,scope=deployboard-api-release,mode=max
    provenance: mode=min
    sbom: true
- name: Record immutable release reference
  env:
    IMAGE: ${{ steps.release.outputs.image }}
    DIGEST: ${{ steps.build.outputs.digest }}
  shell: bash
  run: |
    case "$DIGEST" in
      sha256:*) echo "Published: $IMAGE@$DIGEST" ;;
      *) echo 'No release manifest digest was returned' >&2; exit 1 ;;
    esac
```

Read the jobs as a sequence of authority: gate checks out the revision and tests it; publish waits for that exact gate, verifies that the tag is a simple vMAJOR.MINOR.PATCH string reachable from main, obtains a job-scoped package-write token, and publishes a single-platform image. A tag

does not automatically mean “approved”; configure a protected-tag ruleset and review release authorization separately.

Why GHCR? GitHub Container Registry supports authentication with the job-issued GITHUB_TOKEN and needs no long-lived Docker Hub password. Chapter 25 shows an alternative Docker Hub path; use one registry consistently for the learning release.

26.4 Validate Dockerfile and Compose Inputs

The gate does not start with a green badge; it starts by proving that the files can be interpreted. docker build --check checks Dockerfile build rules and docker compose config -q validates the merged Compose model. Compose resolves file-backed secret paths, so create only disposable QA credentials before that check.

Check from your project root

```
python tools/prepare_qa_secrets.py
docker build --check .
python tools/qa_compose.py config -q
docker build -f Dockerfile.test -t deployboard:ci26-tests .
```

Keep Dockerfile.test separate from the deployable API image. The test image contains dependencies, .github workflow input, the validator, and tests; the runtime Dockerfile excludes these. The root .dockerignore excludes repository history and local secret directories.

FILE Dockerfile.test — complete Chapter 26 test-image revision

```
# A disposable image only for tests. Not a production runtime image.
FROM python:3.13-slim
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
WORKDIR /service
COPY requirements.txt requirements-dev.txt requirements-ci.txt ./
RUN pip install --no-cache-dir -r requirements-dev.txt
COPY .github ./.github
COPY app ./app
COPY migrations ./migrations
COPY alembic.ini ./alembic.ini
COPY tests ./tests
COPY review ./review
COPY tools ./tools
CMD ["python", "-m", "pytest", "-q", "-m", "not stack", "tests"]
```

FILE requirements-ci.txt and amended requirements-dev.txt

```
# requirements-ci.txt
PyYAML==6.0.3

# requirements-dev.txt
-r requirements.txt
pytest==9.0.2
httpx==0.28.1
-r requirements-ci.txt
```

26.5 Run Application Tests in CI

The fast tests exercise authenticated routes, validation, PostgreSQL-like persistence behavior through a disposable SQLite fixture, secret-file handling, performance utilities, image review, and the CI policy. They do not pretend to prove Redis and PostgreSQL work together. The second gate phase starts the actual database, queue, API, publisher, and worker, applies Alembic migrations, runs service-level tests, recreates the API, verifies a preserved record, and checks a deliberately nonzero exit path.

Run the fast container suite (no network or registry needed at runtime)

```
docker run --rm --network none deployboard:ci26-tests   python -m pytest -q -m "not stack" tests
```

Expected: pytest reports a passing test suite and exits 0. A failing test or import error must stop CI with a nonzero code. The full Compose quality gate is in tools/quality_gate.py from Chapter 21; it runs on disposable volumes under the dedicated deployboard-qa21 project.

26.6 Reproduce the CI Quality Gate Locally Without GitHub

The local reproduction is the core learning path. A GitHub account is not needed: the exact same entry point runs inside the hosted gate job and on your machine. The local validator needs only PyYAML; containerized tests install their dependencies from the test image.

Linux, macOS, or Windows with Python available as python

```
python -m venv .ci-venv
# Linux/macOS:
.ci-venv/bin/python -m pip install -r requirements-ci.txt
.ci-venv/bin/python tools/ci26_validate.py
# Windows PowerShell equivalents:
# .\.ci-venv\Scripts\python -m pip install -r requirements-ci.txt
# .\.ci-venv\Scripts\python tools/ci26_validate.py
```

```
python tools/quality_gate.py --cleanup
```

The final command requires a live Docker Engine and may download images the first time. The script exits nonzero on unexpected failures. --cleanup stops its QA containers but intentionally preserves QA named volumes, so persistence checks remain honest; it never removes the everyday DeployBoard project. The GitHub-hosted runner is ephemeral after its workflow finishes.

Do not hide failures If a test reports HOLD, fix the underlying failure before retrying. Do not append || true or replace the exit status with echo SUCCESS.

26.7 Configure Buildx and Reuse Build Cache

The publish job sets up Buildx and uses the GitHub Actions cache backend, scoped to deployboard-api-release. Cache is a speed optimization, not an authenticity claim: BuildKit still resolves the Dockerfile steps and content hashes. Cached results need scrutiny if a build is expected to refresh a remote package or an unpinned base image.

Compare a clean local Buildx build without publishing

```
docker buildx version
docker buildx build --load   --tag deployboard:ci26-local --file Dockerfile .
docker image inspect deployboard:ci26-local   --format '{{.Id}}'
```

The above local --load command loads an image into the Docker Engine; it does not publish it or guarantee that registry attestations are stored. The type=gha cache exporter is a GitHub Actions runner feature, not something to expect from a generic terminal. For more accurate repeat-build timing, use the Chapter 24 benchmark method under comparable cache states.

26.8 Build and Tag Images from Verified Source

The release job checks out the tag revision, validates its semantic name, confirms the commit is reachable from main, and builds the image only after the gate passes. It applies a human-friendly version tag and a second tag containing the source commit SHA. Neither tag is an immutable deployment identity: record the digest returned by the build-push action.

Illustrative release relationship — values are placeholders, not evidence

```
Git tag:      v1.2.3
```

```
Registry tag:  ghcr.io/OWNER/deployboard:1.2.3
Commit tag:    ghcr.io/OWNER/deployboard:sha-<40-hex-commit>
Immutable ref: ghcr.io/OWNER/deployboard@sha256:<64-hex-digest>
```

The exact digest depends on the built manifest and platform. Do not copy the placeholder or interpret the local image ID as a published registry manifest digest. Test the digest pull using the Chapter 25 verification utility after a real release.

26.9 Authenticate Safely and Publish on Trusted Events

The publish job authenticates to ghcr.io using GitHub's short-lived GITHUB_TOKEN with packages:write and contents:read only. Ordinary pushes, pull requests, and workflow_dispatch do not execute that job. The repository/package policies must actually allow publishing. A private package may still require explicit package-access configuration for consumption by another repository.

Configure the repository before your first actual release

- Enable GitHub Actions in the private repository and allow the selected maintained actions.
- Protect the main branch with required checks and apply a tag ruleset for v* releases.
- Confirm the package namespace and visibility; choose the minimal access required by consumers.
- Create the first tag only after merging a verified commit; review the trigger and job permissions.
- Never print GITHUB_TOKEN, database passwords, or a complete environment dump in logs.

Example release commands after code review and required checks

```
git checkout main
git pull --ff-only
git tag -a v1.2.3 -m "DeployBoard tested release 1.2.3"
git push origin v1.2.3
```

These commands are optional: they publish a tag to a GitHub remote, not a container image directly. The workflow then tests that revision and, if

authorized, publishes to GHCR. If release controls are not in place, stop at the local checkpoint.

26.10 Set Minimal Workflow and Token Permissions

The policy validator catches easy-to-miss workflow changes before they can make an unsafe job appear legitimate. It rejects privileged pull-request triggers, broad default permissions, a publish job with no gate dependency, a release condition lacking tag restrictions, and registry secrets used by the gate. It deliberately checks this book's defined workflow, not arbitrary GitHub Actions syntax.

FILE tools/ci26_validate.py — complete listing

```
"""Chapter 26: fail-closed static checks for this teaching workflow.
This is a targeted policy check, not a substitute for GitHub's workflow parser.
Install requirements-ci.txt before running.
"""
from pathlib import Path
import sys
import yaml

ROOT = Path(__file__).resolve().parents[1]
FILE = ROOT / '.github' / 'workflows' / 'deployboard-ci.yml'


def check(workflow):
    if not isinstance(workflow, dict):
        raise ValueError('Workflow must be a YAML mapping')
    # PyYAML treats the YAML 1.1 key 'on' as True.
    triggers = workflow.get('on', workflow.get(True))
    if not isinstance(triggers, dict):
        raise ValueError('Missing event triggers')
    if 'pull_request_target' in triggers or 'workflow_run' in triggers:
        raise ValueError('Untrusted privileged triggers are forbidden')
    for name in ('pull_request', 'push'):
        if name not in triggers:
            raise ValueError('Missing required event: ' + name)
    push = triggers['push']
    if (not isinstance(push, dict) or
            push.get('branches') != ['main'] or
            push.get('tags') != ['v*']):
        raise ValueError('Pushes must be restricted to main and version tags')
    if workflow.get('permissions') != {'contents': 'read'}:
        raise ValueError('Default token must be contents:read only')
    jobs = workflow.get('jobs', {})
```

```
    if not isinstance(jobs, dict) or not {'gate', 'publish'} <= jobs.keys():
        raise ValueError('Missing isolated gate or publish job')
    gate, release = jobs['gate'], jobs['publish']
    if 'permissions' in gate and gate['permissions'] != {'contents': 'read'}:
        raise ValueError('Gate has excess token permissions')
    if release.get('needs') != ['gate']:
        raise ValueError('Publish must depend on the quality gate')
    if release.get('permissions') != {'contents': 'read', 'packages': 'write'}:
        raise ValueError('Publish job has incorrect token permissions')
    condition = release.get('if', '')
    if not all(fragment in condition for fragment in
               ('github.event_name', "'push'", 'refs/tags/v')):
        raise ValueError('Publishing must be restricted to version-tag pushes')
    gate_steps = gate.get('steps', [])
    if not any('ci26_validate.py' in str(s.get('run', '')) for s in gate_steps):
        raise ValueError('Gate must validate its workflow')
    if not any('quality_gate.py --cleanup' in str(s.get('run', ''))
               for s in gate_steps):
        raise ValueError('Gate must execute the container quality gate')
    if '${{ secrets.' in str(gate_steps):
        raise ValueError('Untrusted gate must not receive registry secrets')
    release_steps = release.get('steps', [])
    builds = [s for s in release_steps
              if 'docker/build-push-action@' in s.get('uses', '')]
    if len(builds) != 1 or builds[0].get('with', {}).get('push') is not True:
        raise ValueError('Missing explicit registry publish step')
    if not any('git merge-base --is-ancestor' in str(s.get('run', ''))
               for s in release_steps):
        raise ValueError('Release ancestry must be verified')
    if not any('docker/login-action@' in s.get('uses', '')
               for s in release_steps):
        raise ValueError('Missing registry login action')
    if (not (ROOT/'Dockerfile').is_file() or
            not (ROOT/'compose.qa.yaml').is_file()):
        raise ValueError('Required Docker and QA files are absent')
    return True

def main():
    try:
        data = yaml.safe_load(FILE.read_text(encoding='utf-8'))
        check(data)
    except (OSError, ValueError, yaml.YAMLError) as exc:
        sys.exit('HOLD: CI configuration validation failed: ' + str(exc))
    print('PASS: DeployBoard CI event, permissions, and gate checks')
    print('NOTE: Pin verified action SHAs or a lockfile in production.')

if __name__ == '__main__':
```

```
    main()
```

FILE tests/test_ci26.py — complete policy-regression suite

```
"""Focused Chapter 26 policy tests. Run: python -m pytest -q tests/test_ci26.py"""
from copy import deepcopy
from pathlib import Path
import yaml
import pytest
from tools.ci26_validate import check

ROOT = Path(__file__).resolve().parents[1]
DATA = yaml.safe_load((ROOT/'.github/workflows/deployboard-ci.yml').read_text())

def test_known_workflow_passes():
    assert check(DATA)

def test_privileged_pr_trigger_is_rejected():
    candidate = deepcopy(DATA)
    candidate[True]['pull_request_target'] = {}
    with pytest.raises(ValueError, match='Untrusted'):
        check(candidate)

def test_broad_default_permissions_are_rejected():
    candidate = deepcopy(DATA)
    candidate['permissions'] = {'contents': 'write'}
    with pytest.raises(ValueError, match='Default token'):
        check(candidate)

def test_publish_must_wait_for_gate():
    candidate = deepcopy(DATA)
    candidate['jobs']['publish']['needs'] = []
    with pytest.raises(ValueError, match='depend'):
        check(candidate)

def test_publish_requires_tag_only_condition():
    candidate = deepcopy(DATA)
    candidate['jobs']['publish']['if'] = "${{ github.event_name == 'push' }}"
    with pytest.raises(ValueError, match='version-tag'):
        check(candidate)

def test_publish_write_scope_is_narrow():
    candidate = deepcopy(DATA)
    candidate['jobs']['publish']['permissions']['contents'] = 'write'
    with pytest.raises(ValueError, match='incorrect'):
        check(candidate)

def test_gate_has_no_registry_secrets():
```

```
candidate = deepcopy(DATA)
candidate['jobs']['gate']['steps'].append({
    'run': 'echo ${{ secrets.DOCKER_PASSWORD }}'
})
with pytest.raises(ValueError, match='registry secrets'):
    check(candidate)
```

Limit of static checks The validator does not authenticate action publishers, prove an action SHA is trusted, inspect transitive dependencies, or replace GitHub's server-side validation. Before production, prefer immutable action revisions or workflow lockfiles and review organization settings.

26.11 Inspect Build Summaries and Failed Runs

Open the workflow run in GitHub Actions and read the result in execution order: workflow input validation → isolated QA → publish tag check → Buildx build → registry digest. The build-push action can attach a build summary and build record; treat them as evidence to review, not as proof the application is deployable.

Symptom	Likely cause	First check
Validator fails	Changed permissions, triggers or job dependency	Run tools/ci26_validate.py locally
QA fails at config	Missing disposable secret files or bad Compose overlay	Run prepare_qa_secrets, then config -q
Tests fail after import	Test image missing CI requirements or .github input	Rebuild Dockerfile.test without cached assumptions
Tag job skipped	Event is not a trusted version-tag push	Inspect event_name and ref; skip is correct for PR
GHCR push denied	Job permission, package setting or organization policy	Check job packages:write and GHCR policy
Build cache unavailable	Cache quota, scope, backend or runner limitation	Build cleanly; do not bypass tests

When a run fails, record the failing step, checked-out SHA, non-secret log excerpt, expected outcome, and change made before re-running. Never "fix" a release by allowing a test failure or moving the tag without review.

26.12 Checkpoint: Produce a Tested Image Automatically

Your core checkpoint is complete without GitHub when the workflow-policy check passes, the Dockerfile and Compose inputs validate, the isolated Compose gate succeeds, and a local image builds. For the optional online checkpoint, open a pull request to prove that publishing is skipped, then release a protected vX.Y.Z tag from main, verify the GHCR digest, and retrieve that exact digest elsewhere as taught in Chapter 25.

Release acceptance record — fill in actual observations

Evidence	Record or expected result
Source revision	Full Git commit SHA: _____
Local policy validator	PASS / HOLD: _____
Isolated Compose gate	PASS / HOLD; run date: _____
Pull-request publish job	Skipped: YES / NO (must be YES)
Protected release tag	v____.____.____ reviewer: _____
Published digest	sha256: _____
Independent digest pull	PASS / HOLD; host or runner: _____

Release decision APPROVE only with verified evidence; otherwise HOLD. A successful image build is not sufficient if runtime checks, protection rules, or the digest record are missing.

Carry this gate forward into Chapter 27, where automation will publish release artifacts with an explicit promotion decision rather than granting every passing branch a deployment credential.

Chapter 27 — Prepare a Linux Server for Reliable Docker Deployment

The release image you built and approved in Chapters 25–26 needs a trustworthy place to run. In this chapter you prepare that place before moving any real application records or opening a public port. You can follow the entire core path in a free local Ubuntu virtual machine; a remote Linux server is optional.

DELIVERABLE A reviewed Ubuntu host, a separate deployment-candidate Compose file, private credential staging, verified Docker tooling, and a signed-off go/no-go worksheet. The public release happens later—not in this host-preparation chapter.

SAFE DEFAULT The application proxy listens on 127.0.0.1:8080 only; Redis, PostgreSQL, and the Docker API never publish host ports. A local VM needs no DNS name or paid cloud service.

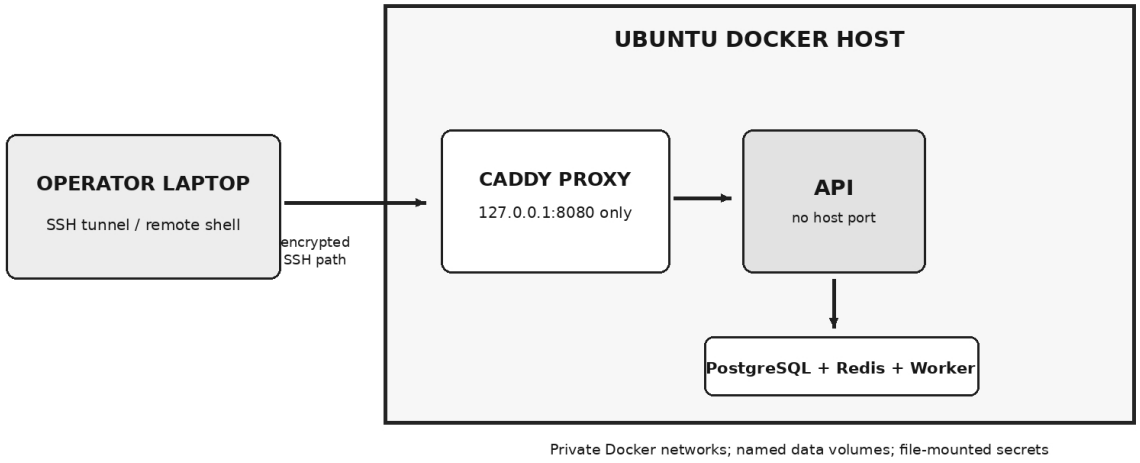


Figure 27.1 The host boundary: only an SSH tunnel or local session reaches the unpublished DeployBoard candidate.

27.1 Choose a Free Local Linux VM or an Optional Remote Server

Choose Ubuntu Server 24.04 LTS (amd64 or arm64) as the reference Linux host. The free learning path is a local VirtualBox, Hyper-V, or similar VM. Select NAT networking initially; port forwarding or SSH can be added after you understand the guest address. Keep a hypervisor snapshot before installing packages or changing SSH settings.

Environment	Best use	Boundary / caution
Local Linux VM	All core installation, storage and preflight labs	Free software path; no public IP required
Spare Linux machine	Repeatable dedicated-host exercises	May expose services to your LAN; review router rules
Optional remote VPS	Remote SSH, upstream firewall and external exposure tests	Real charges, provider network rules and credential risk

On a remote host, provision an ordinary non-root administrator with sudo access. Record your operating-system version and CPU architecture before choosing published image platforms. Do not treat a cloud firewall as equivalent to Docker container isolation.

27.2 Estimate Host Resources, Storage, and Ongoing Costs

Resource	Learning-host starting point	Why it matters
CPU	2 vCPU	API, migration, proxy and background tasks
RAM	4 GiB (3 GiB minimum preflight)	PostgreSQL + Redis + worker can exhaust a tiny host
Disk	40 GiB virtual disk, 15 GiB free at check time	Images, build cache, volumes, logs, backups
Network	Outbound HTTPS; SSH for remote admin	Package and image pulls, verified administration

These are planning estimates, not guarantees or live measurements. Watch disk consumption with df -h, Docker footprint with docker system df, and memory with free -h. For an optional VPS, budget monthly as (compute × hours) + storage + snapshot/backup + outbound transfer + taxes. Look up current rates and regional transfer rules before buying.

RUN ON THE UBUNTU HOST — OBSERVE, DO NOT MODIFY

```
uname -m
cat /etc/os-release
free -h
df -h /
lsblk -f
```

27.3 Install and Verify Docker Engine on the Server

Install from Docker's official Ubuntu apt repository, not an unreviewed convenience script. Do this on a clean Ubuntu host; if Docker packages already exist, review the official conflict-removal procedure first instead of blindly uninstalling anything that may own data.

ON UBUNTU — OFFICIAL DOCKER APT REPOSITORY

```
sudo apt update
sudo apt install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL \
  https://download.docker.com/linux/ubuntu/gpg \
  -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
sudo tee /etc/apt/sources.list.d/docker.sources >/dev/null <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
sudo apt update
sudo apt install -y docker-ce docker-ce-cli \
  containerd.io docker-buildx-plugin docker-compose-plugin
```

EXPECTED CHECK — NOT A FABRICATED RESULT

```
sudo systemctl is-active docker
sudo docker version
sudo docker compose version
sudo docker run --rm hello-world
```

The daemon must be active, Compose must report a version, and the hello-world container should exit after printing its welcome confirmation. If apt cannot identify the Ubuntu codename, verify that the distribution is supported and re-check the official install page. Record exact installed versions in your evidence worksheet.

27.4 Secure SSH, Host Firewalls, and Docker Port-Publishing Rules

For a remote server, install your SSH public key, confirm that a second terminal can log in using that key, and only then consider disabling password authentication or direct root login. Keep the existing session open while validating and reloading SSH. An incorrect change can lock you out of the server.

VERIFY BEFORE CHANGING SSH OR FIREWALL RULES

```
sudo sshd -t
sudo systemctl status ssh --no-pager
sudo ss -lnt
```

Keep SSH open only from trusted addresses where feasible, through the provider security group and host firewall. Allowing a port in UFW is not a reliable way to restrict a Docker-published service: Docker manipulates NAT/filtering chains and published traffic may bypass UFW's usual INPUT policy. The safe learning pattern uses 127.0.0.1 binding plus an external verification probe, not a presumed UFW deny.

DO NOT DO THIS Do not publish the Docker daemon on tcp://0.0.0.0:2375, disable Docker firewall integration for convenience, or enable a public API port before TLS and access controls are planned.

27.5 Handle Docker Group Privileges and Daemon Exposure

The rootful Docker daemon can create privileged containers and mount host paths. Membership in the docker group is therefore effectively root-equivalent, not an ordinary application-user permission. Prefer sudo docker for host administration while learning; use a tightly controlled administrator account, strong SSH authentication, and audit logs.

INSPECT THE DAEMON ACCESS BOUNDARY

```
id
getent group docker || true
ls -l /var/run/docker.sock
sudo docker info --format "{{.ServerVersion}}"
```

If you later choose rootless Docker, test its networking, low-port binding, cgroup limits and storage paths separately. A remote Docker context

connects to the remote daemon but does not reduce the privileges of an account permitted to control it.

27.6 Plan DNS, Ports, and Network Boundaries

Reserve port 22 for tightly controlled SSH administration. Keep DeployBoard on localhost port 8080 in this chapter. Do not publish PostgreSQL 5432, Redis 6379, or API 8000 to the host. Record planned DNS name, public HTTPS certificate issuer, ports 80/443 and rotation/renewal responsibility as future release decisions—not as permissions to open them now.

Endpoint	Current host policy	Verification
SSH (22/tcp)	Remote administration only; restrict sources	Two-terminal SSH test + upstream firewall rules
Caddy (8080/tcp)	127.0.0.1 only	Local curl succeeds; external TCP probe fails
API (8000/tcp)	Private edge network only	No host-published port
PostgreSQL / Redis	Private internal Docker network	No host-published ports or public route
Docker socket	UNIX socket only	No unauthenticated TCP listener

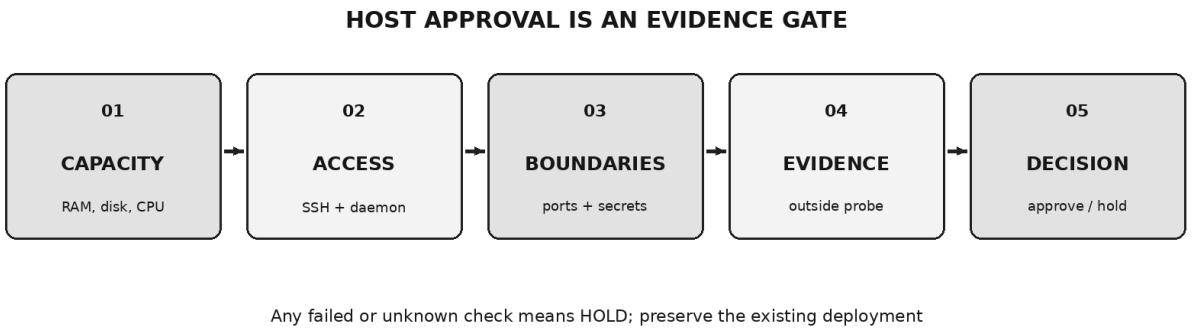


Figure 27.2 Host approval follows evidence; an unknown or failed boundary is a release hold.

27.7 Separate Development and Production Compose Configuration

The Chapter 26 archive inherits a Compose project named deployboard-ch23-sandbox from the incident lab. Do not rename it in place and assume its named volumes follow. The host-candidate configuration below deliberately uses deployboard-host27, a new project with empty, independent volumes. Any existing real data requires a documented PostgreSQL backup and restore, covered in the later deployment and recovery chapters.

Unlike the developer configuration, the host file uses images from an immutable registry digest and contains no build: instruction. The API, publisher, worker and migration task use the same digest. The proxy is localhost-only, and internal services are never published. Copy the following complete file into the root of the DeployBoard project.

COMPLETE FILE — compose.host27.yaml

```
# Chapter 27: independent DEPLOYMENT-HOST CANDIDATE, not a modification of
# the Chapter 23 troubleshooting stack. No public application ports yet.
name: deployboard-host27
x-app-env: &app_env
  DB_HOST: db
  DB_NAME: deployboard
  DB_USER: deployboard
  DB_PASSWORD_FILE: /run/secrets/db_password
  REDIS_HOST: redis
x-app-safety: &app_safety
  read_only: true
  cap_drop: [ALL]
  security_opt: ["no-new-privileges:true"]
  tmpfs: ["/tmp:rw,nosuid,noexec,size=32m,mode=1777"]
  pids_limit: 128
x-logs: &service_logs
  driver: local
  options: {max-size: "10m", max-file: "3"}
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_DB: deployboard
      POSTGRES_USER: deployboard
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets: [db_password]
    volumes: ["dbdata:/var/lib/postgresql/data"]
    networks: [private]
    logging: *service_logs
    restart: unless-stopped
```

```
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
      interval: 5s
      timeout: 3s
      retries: 12
      start_period: 15s
  redis:
    image: redis:7.4.11-bookworm
    command: ["redis-server", "--appendonly", "yes", "--appendfsync", "everysec"]
    volumes: ["redisdata:/data"]
    networks: [private]
    logging: *service_logs
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 12
  api:
    <<: *app_safety
    image: ${DEPLOYBOARD_IMAGE:?Set a verified registry image by digest}
    environment:
      <<: *app_env
      DEPLOYBOARD_VIEWER_TOKEN_FILE: /run/secrets/viewer_token
      DEPLOYBOARD_OPERATOR_TOKEN_FILE: /run/secrets/operator_token
    secrets: [db_password, viewer_token, operator_token]
    depends_on:
      db: {condition: service_healthy}
    networks: [edge, private]
    logging: *service_logs
    restart: unless-stopped
    mem_limit: 384m
    cpus: 1.0
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request;
urllib.request.urlopen('http://127.0.0.1:8000/ready', timeout=3)"]
      interval: 10s
      timeout: 4s
      retries: 3
      start_period: 20s
  publisher:
    <<: *app_safety
    image: ${DEPLOYBOARD_IMAGE:?Set a verified registry image by digest}
    command: ["python", "-m", "app.publisher"]
    environment: *app_env
    secrets: [db_password]
    depends_on:
      db: {condition: service_healthy}
```

```
      redis: {condition: service_healthy}
    networks: [private]
    logging: *service_logs
    restart: unless-stopped
    mem_limit: 256m
    cpus: 0.5
  worker:
    <<: *app_safety
    image: ${DEPLOYBOARD_IMAGE:?Set a verified registry image by digest}
    command: ["python", "-m", "app.worker"]
    environment: *app_env
    secrets: [db_password]
    depends_on:
      db: {condition: service_healthy}
      redis: {condition: service_healthy}
    networks: [private]
    logging: *service_logs
    restart: unless-stopped
    mem_limit: 256m
    cpus: 0.5
    stop_grace_period: 15s
  proxy:
    image: caddy:2.11.4-alpine
    ports: ["127.0.0.1:${DEPLOYBOARD_HTTP_PORT:-8080}:8080"]
    volumes:
      - ./proxy/Caddyfile:/etc/caddy/Caddyfile:ro
      - ./site:/srv:ro
    depends_on:
      api: {condition: service_healthy}
    networks: [edge]
    logging: *service_logs
    restart: unless-stopped
    mem_limit: 192m
    cpus: 0.5
    cap_drop: [ALL]
    security_opt: ["no-new-privileges:true"]
  migrate:
    <<: *app_safety
    image: ${DEPLOYBOARD_IMAGE:?Set a verified registry image by digest}
    profiles: [tools]
    command: ["alembic", "upgrade", "head"]
    environment: *app_env
    secrets: [db_password]
    depends_on:
      db: {condition: service_healthy}
    networks: [private]
    logging: *service_logs
networks:
```

```
  edge: {driver: bridge}
  private: {driver: bridge, internal: true}
volumes:
  dbdata:
  redisdata:
secrets:
  db_password: {file: ./private/db_password.txt}
  viewer_token: {file: ./private/viewer_token.txt}
  operator_token: {file: ./private/operator_token.txt}
```

Do not start this configuration with a placeholder image: record and validate the real registry digest from Chapter 25. Run migrations explicitly after storage and secrets are provisioned, and only start the remaining services after the database is healthy and the migration succeeds. Compose depends_on expresses startup checks; it does not guarantee ongoing database failover or a complete rollback policy.

27.8 Prepare Persistent Data Paths and Secret Files

PostgreSQL 17 in this application stores its database cluster at /var/lib/postgresql/data; Redis append-only files live under /data. Named Docker volumes keep these data paths separate from container lifetimes. Plan volume growth, backups, off-host copies and restore testing before moving production records.

FRESH HOST ONLY — PREPARE SECRETS WITHOUT PRINTING CONTENT

```
sudo install -d -m 0700 /opt/deployboard
# Put the extracted source folder in your home directory first.
sudo cp -a ~/DeployBoard_Chapter_27_Source/. /opt/deployboard/
cd /opt/deployboard
sudo python3 tools/prepare_ch19_secrets.py
sudo ls -ld private
sudo find private -maxdepth 1 -type f \
  -printf "%f  %m\n"
```

For a remote host, first copy the extracted folder with: scp -r DeployBoard_Chapter_27_Source ops@server.example.com:~/DeployBoard_Chapter_27_Source (replace the account and host). For a local VM, extract the folder to your Linux home directory. The inherited preparation utility creates three cryptographically random credentials if none exist. On a new host, run it once; do not

overwrite existing secrets. On a host with production data, the database password must match the existing PostgreSQL role, and credential rotation requires a planned database-side change. Host-side protection includes a private directory, restrictive ownership and controlled backups; file-mounted Compose secrets are not encrypted at rest by Compose.

HOST OWNERSHIP The sample copies files to /opt/deployboard but does not automatically assign an operator. Ensure the operator that runs Compose can read project files and secret sources without opening access to ordinary users. A strict 0700 private directory and use of sudo with a non-root service account are compatible with the included file-mounted credentials.

27.9 Connect Through Secure Docker Contexts Where Appropriate

For a remote host, SSH administration is the simplest option: copy the project using scp or rsync, log in, then run sudo docker compose there. This avoids relying on remote bind-mounted paths that have not been staged. An optional Docker SSH context works only when the remote account is already authorized to use the daemon socket.

OPTIONAL — REPLACE HOSTNAME AND ACCOUNT; REMOTE SOCKET ACCESS REQUIRED

```
docker context create deployboard-remote \
  --docker "host=ssh://ops@server.example.com"
docker --context deployboard-remote version
docker context ls
```

A context selects a daemon, not a file-sync mechanism. Local Compose relative bind mounts, Linux filesystem permissions, and secrets must be validated on the host where they are consumed. Do not enable insecure unencrypted Docker TCP access to make remote contexts easier.

27.10 Run Preflight Checks Before Installing the Application

The preflight utility below reads memory, free disk, Linux identity, Docker CLI/daemon and Compose availability. It makes no changes and prints no secret values. A missing Docker permission is a HOLD, not a reason to add every user to the docker group.

COMPLETE FILE — tools/host27_preflight.py

```
"""Read-only Ubuntu/Linux Docker host preflight for Chapter 27.
No secret files opened; no firewall, Docker, or sshd settings changed.
Exit 0 means checks passed, 2 means a deployment hold is required.
"""
import argparse
import json
import platform
from pathlib import Path
import shutil
import subprocess
import sys
MIN_MEMORY_MIB = 3072
MIN_FREE_GIB = 15

def memory_mib(path=Path('/proc/meminfo')):
    try:
        for line in path.read_text().splitlines():
            if line.startswith('MemTotal:'):
                return int(line.split()[1]) // 1024
    except (OSError, ValueError, IndexError):
        pass
    return None

def command(args, timeout=8):
    try:
        p = subprocess.run(args, text=True, capture_output=True,
                           timeout=timeout, check=False)
        return p.returncode == 0, (p.stdout or p.stderr).strip()[:200]
    except (OSError, subprocess.TimeoutExpired):
        return False, 'unavailable or timed out'

def check(root=Path('/')):
    checks = []
    def add(name, passed, explanation):
        checks.append({'name': name, 'pass': bool(passed), 'detail': explanation})
    linux = platform.system() == 'Linux'
    add('Linux host', linux, platform.system())
    mem = memory_mib()
    add('Memory >= 3 GiB', mem is not None and mem >= MIN_MEMORY_MIB,
        f'{mem} MiB reported' if mem is not None else 'unavailable')
    try:
        free_gib = shutil.disk_usage(root).free / (1024 ** 3)
    except OSError:
        free_gib = 0.0
    add('Disk free >= 15 GiB', free_gib >= MIN_FREE_GIB,
        f'{free_gib:.1f} GiB free on {root}')
```

```
    add('SSH client (remote option)', True,
        'installed' if shutil.which('ssh') else 'optional on a local VM')
    if not shutil.which('docker'):
        add('Docker Engine CLI', False, 'Install from official apt repository')
        add('Docker daemon accessible', False, 'CLI absent')
        add('Compose plugin', False, 'CLI absent')
    else:
        add('Docker Engine CLI', True, 'docker found on PATH')
        ok, detail = command(['docker', 'version', '--format',
                              '{{.Server.Version}}'])
        add('Docker daemon accessible', ok, detail or 'no response')
        ok, detail = command(['docker', 'compose', 'version', '--short'])
        add('Compose plugin', ok, detail or 'no response')
    return checks

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--json', action='store_true', help='machine-readable results')
    args = parser.parse_args()
    checks = check()
    if args.json:
        print(json.dumps({'checks': checks, 'approved': all(x['pass'] for x in checks)},
                         indent=2))
    else:
        for item in checks:
            print(('PASS' if item['pass'] else 'HOLD') + ': ' + item['name']
                  + ' — ' + item['detail'])
        print('No firewall, credentials, data, or Docker settings were changed.')
    return 0 if all(x['pass'] for x in checks) else 2

if __name__ == '__main__':
    sys.exit(main())
```

RUN ON THE ACTUAL UBUNTU HOST

```
sudo python3 tools/host27_preflight.py
# Optional machine-readable output:
sudo python3 tools/host27_preflight.py --json
```

Each PASS is an observed check on that host; every HOLD must be resolved before deployment. The 3 GiB / 15 GiB thresholds are reference minima for the learning stack, not production sizing guarantees. Capture the result along with hostname, Ubuntu release, Docker/Compose versions and the expected image digest.

The following validator enforces the host-only port policy and a real name@sha256:… reference. It cannot certify the remote registry; Chapter 25 digest and scan evidence still apply.

COMPLETE FILE — tools/host27_release.py

```
"""Validate immutable image reference and host Compose boundaries offline."""
import argparse
from pathlib import Path
import re
import sys
import yaml
DIGEST = re.compile(r'^[A-Za-z0-9._:/-]+@sha256:[0-9a-f]{64}$')

def check_image(image):
    return bool(DIGEST.fullmatch(image or '')) and 'example.invalid' not in image

def check_compose(path):
    cfg = yaml.safe_load(Path(path).read_text(encoding='utf-8'))
    problems = []
    services = cfg['services']
    if cfg.get('name') == 'deployboard-ch23-sandbox':
        problems.append('Host Compose reuses Chapter 23 sandbox project name')
    for name in ('db', 'redis', 'api', 'worker', 'publisher', 'migrate'):
        if 'ports' in services[name]:
            problems.append(f'{name} publishes a host port')
    for name in ('api', 'worker', 'publisher', 'migrate'):
        if 'build' in services[name]:
            problems.append(f'{name} builds on deployment host')
    for published in services['proxy'].get('ports', []):
        if not str(published).startswith('127.0.0.1:'):
            problems.append('proxy must bind only to 127.0.0.1 in Chapter 27')
    if not cfg['networks']['private'].get('internal'):
        problems.append('private network must be internal')
    return problems

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--image', required=True, help='actual immutable registry image digest')
    parser.add_argument('--compose', default='compose.host27.yaml')
    args = parser.parse_args()
    problems = check_compose(args.compose)
    if not check_image(args.image):
        problems.append('DEPLOYBOARD_IMAGE must be a real name@sha256:<64 hex>')
    if problems:
        for p in problems:
            print('HOLD:', p)
        return 2
    print('PASS: image reference and host-only Compose boundaries validated')
    print('Requires separate live Docker/registry/network proof before deployment.')
    return 0
```

```
if __name__ == '__main__':
    sys.exit(main())
```

AFTER REPLACING THE IMAGE WITH THE ACTUAL 64-HEX DIGEST

```
python3 -m pip install -r requirements-ci.txt
export
DEPLOYBOARD_IMAGE="ghcr.io/YOUR_ORG/deployboard@sha256:REAL_DIGEST"
python3 tools/host27_release.py --image "$DEPLOYBOARD_IMAGE"
sudo env DEPLOYBOARD_IMAGE="$DEPLOYBOARD_IMAGE" \
  docker compose -f compose.host27.yaml config -q
```

EXPECTED An example containing REAL_DIGEST must fail until replaced. No valid digest or registered registry account is supplied by the book. A successful config -q shows the Compose configuration parses; it does not start containers or validate image authenticity.

27.11 Test External Port Exposure from Outside the Host

A self-test from the Docker host cannot establish whether a service is publicly reachable. First check the bound socket locally, then test from an independent device or outside network. For a local-only VM, adapt this to a separate LAN machine and its firewall boundary; an Internet reachability test is optional.

RUN AFTER A FUTURE VERIFIED LOCALHOST-BOUND START

```
sudo ss -lntp
curl -fsS http://127.0.0.1:8080/health
```

The curl request should succeed only after the application has actually started. Before deployment it will ordinarily fail; do not mistake that for a failed host preflight. The external script tests one chosen TCP endpoint and treats any successful connection as a release HOLD.

COMPLETE FILE — tools/host27_exposure.py

```
"""Check unintended TCP exposure FROM A SEPARATE EXTERNAL MACHINE.
Tests only the IP/port and vantage point supplied; cannot prove internet-wide
inaccessibility. This script never scans port ranges.
"""
import argparse
import socket
import sys

def reachable(host, port, timeout=3.0):
```

```
    try:
        infos = socket.getaddrinfo(host, port, type=socket.SOCK_STREAM)
    except socket.gaierror as exc:
        raise ValueError(f'DNS lookup failed: {exc}') from exc
    for family, socktype, proto, _, addr in infos:
        try:
            with socket.socket(family, socktype, proto) as sock:
                sock.settimeout(timeout)
                if sock.connect_ex(addr) == 0:
                    return True
        except OSError:
            continue
    return False

def main():
    p = argparse.ArgumentParser()
    p.add_argument('--host', required=True, help='actual remote host IP or DNS name')
    p.add_argument('--port', required=True, type=int)
    a = p.parse_args()
    if not (1 <= a.port <= 65535):
        p.error('port must be 1..65535')
    if a.host in ('localhost', '127.0.0.1', '::1'):
        p.error('use an external address and execute on a different machine')
    try:
        exposed = reachable(a.host, a.port)
    except ValueError as exc:
        print(f'INCONCLUSIVE: {exc}')
        return 2
    if exposed:
        print(f'HOLD: TCP {a.host}:{a.port} reachable from this client')
        return 1
    print(f'OBSERVED CLOSED/FILTERED: {a.host}:{a.port} from THIS client')
    print('Repeat from a separate network and inspect Docker publish rules.')
    return 0

if __name__ == '__main__':
    sys.exit(main())
```

RUN FROM A DIFFERENT NETWORK — REPLACE THE HOST

```
python3 tools/host27_exposure.py \
  --host YOUR_SERVER_PUBLIC_IP --port 8080
```

Expected outcome for the default host configuration: CLOSED/FILTERED from the external vantage point. A timeout proves only what this client observed, not worldwide inaccessibility; compare it with docker compose ps, ss, upstream cloud-firewall rules, and independent second-network checks. Never probe unrelated hosts.

27.12 Checkpoint: Approve the Deployment Host

Run the six non-destructive offline validation tests supplied with this chapter. They check that the Compose file preserves private services, the image policy rejects mutable or placeholder releases, and the external port probe correctly detects an unexpected open port.

COMPLETE FILE — tests/test_host27.py

```
"""Offline checks for the Chapter 27 safety tooling."""
import tempfile
from pathlib import Path
from unittest import mock
import pytest
from tools import host27_preflight as preflight
from tools import host27_release as release
from tools import host27_exposure as exposure

def test_image_digest_rejects_tag_placeholder_and_uppercase():
    good = 'ghcr.io/example/deployboard@sha256:' + 'a' * 64
    assert release.check_image(good)
    for bad in ['deployboard:latest', 'ghcr.io/example/x@sha256:123',
                'registry.example.invalid/deployboard@sha256:' + 'a' * 64]:
        assert not release.check_image(bad)

def test_host_compose_safe():
    assert release.check_compose(Path(__file__).parents[1] / 'compose.host27.yaml') == []

def test_host_compose_detects_public_proxy():
    p = Path(__file__).parents[1] / 'compose.host27.yaml'
    with tempfile.TemporaryDirectory() as d:
        f = Path(d) / 'candidate.yaml'
        f.write_text(p.read_text().replace('127.0.0.1:${DEPLOYBOARD_HTTP_PORT:-8080}',
                                           '0.0.0.0:${DEPLOYBOARD_HTTP_PORT:-8080}'))
        assert 'proxy must bind only to 127.0.0.1 in Chapter 27' in release.check_compose(f)

def test_meminfo_parser():
    with tempfile.TemporaryDirectory() as d:
        f = Path(d) / 'meminfo'
        f.write_text('MemTotal:       4194304 kB\n')
        assert preflight.memory_mib(f) == 4096

def test_external_probe_reachable(monkeypatch):
    fake_socket = mock.MagicMock()
    fake_socket.connect_ex.return_value = 0
    fake_socket.__enter__.return_value = fake_socket
    monkeypatch.setattr(exposure.socket, 'getaddrinfo',
```

```
                lambda h, p, type: [(2, 1, 0, '', (h, p))])
    monkeypatch.setattr(exposure.socket, 'socket', lambda *args: fake_socket)
    assert exposure.reachable('203.0.113.10', 8080)

def test_external_probe_not_reachable(monkeypatch):
    fake_socket = mock.MagicMock()
    fake_socket.connect_ex.return_value = 111
    fake_socket.__enter__.return_value = fake_socket
    monkeypatch.setattr(exposure.socket, 'getaddrinfo',
                lambda h, p, type: [(2, 1, 0, '', (h, p))])
    monkeypatch.setattr(exposure.socket, 'socket', lambda *args: fake_socket)
    assert not exposure.reachable('203.0.113.10', 8080)
```

CHECKPOINT COMMANDS — DO NOT START OR DELETE ANY VOLUMES

```
python3 -m pytest -q tests/test_host27.py
# After installing Docker and creating actual credentials:
sudo python3 tools/host27_preflight.py
python3 tools/host27_release.py --image "$DEPLOYBOARD_IMAGE"
sudo env DEPLOYBOARD_IMAGE="$DEPLOYBOARD_IMAGE" \
  docker compose -f compose.host27.yaml config -q
```

Evidence	Required proof	Decision
Host suitability	Linux, memory, disk, Compose, Engine recorded	PASS / HOLD
Privileges	Only trusted administrators control Docker socket	PASS / HOLD
Connectivity	SSH verified without lockout; external 8080 blocked	PASS / HOLD
Configuration	Real immutable digest; no build or database ports	PASS / HOLD
Data and secrets	New volumes or approved restore; protected secret sources	PASS / HOLD
Release readiness	Quality gate, image scan and registry evidence attached	PASS / HOLD

RELEASE RECORD Host identifier: _____ Ubuntu release: _____ Docker: _____ Compose: _____ Image digest: _____ Verified externally from: _____ Reviewer/date: _____ APPROVE / HOLD: _____

If any result is missing, record HOLD and correct the host rather than weakening isolation. The end state of Chapter 27 is a suitable and reviewed

deployment host—not a publicly exposed application. Next, deploy the approved image with domain, HTTPS, rollout and rollback safeguards.

Chapter 28 — Deploy DeployBoard with HTTPS and Controlled CI/CD

From an approved Linux host to a verifiable release: keep the data, publish only HTTPS, prove who can write, and preserve a recovery point.

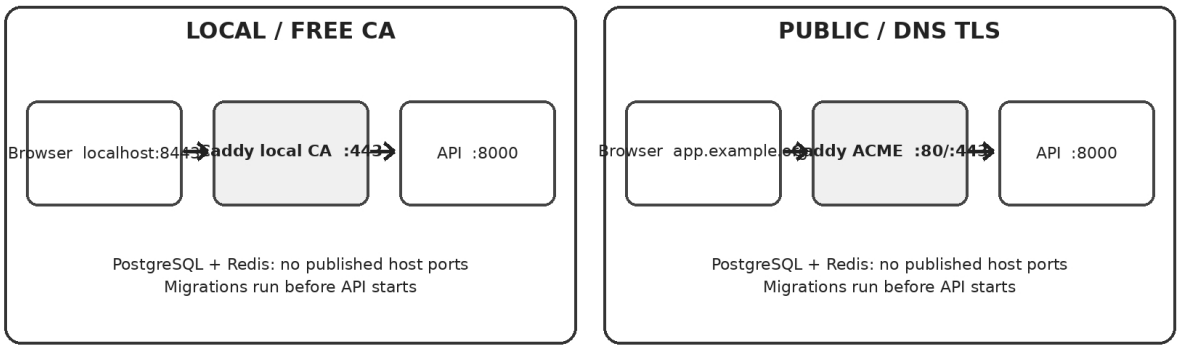
Mission Deploy the same FastAPI, PostgreSQL, Redis, publisher, worker, and Caddy stack developed in Chapters 13–27. Use an immutable registry image, perform schema changes deliberately, and retain evidence that the service works after the change.

You will produce A secure local HTTPS deployment, an optional DNS-backed public deployment, a gated GitHub Actions deploy job, TLS/API smoke evidence, a database backup, and a release manifest. No fabricated digest, live certificate, or approval result is supplied.

Prerequisites Chapter 27 host approved; Docker Engine and Compose installed; a real Chapter 28 image already built and published from the updated source; host-managed credential files in private/; approximately 3 GiB RAM and 15 GiB free disk for the learning VM.

IMPORTANT • The lab uses a new Compose project, deployboard-prod28. Do not rename the project midway through deployment: doing so changes the named-volume identities and makes existing records appear missing. Never run docker compose down -v on persistent workloads.

FIGURE 28.1 | TWO HTTPS RELEASE BOUNDARIES



The local route uses a Caddy-managed internal CA. The public route requires DNS and a public CA. Both terminate TLS at the proxy; PostgreSQL and Redis remain private.

28.1 Choose a Production Image by Immutable Digest

The output of Chapter 26 is not a production release until you can name the exact registry manifest to pull. Tags such as 1.2.3 are useful human labels but can be repointed. A digest ties the deployment to a specific registry object. Build a new image after adding Chapter 28's application-side rate limiter; do not reuse a Chapter 27 digest for source it does not contain.

Establish the immutable reference

LISTING • Terminal — release identity (5 lines)

```
# Replace OWNER, REPO and the digest with YOUR published values.
export DEPLOYBOARD_IMAGE='ghcr.io/OWNER/REPO@sha256:ACTUAL_64_HEX_DIGEST'
docker buildx imagetools inspect "$DEPLOYBOARD_IMAGE"
docker pull "$DEPLOYBOARD_IMAGE"
docker image inspect "$DEPLOYBOARD_IMAGE" --format '{{json .RepoDigests}}'
```

A digest proves image identity, not that its dependencies are safe. Carry forward the scan review, source commit, and approval record from Chapters 20 and 25. Multi-platform manifests may point to distinct platform-specific image digests; record the exact index digest you deploy and the target architecture.

Stop condition Do not substitute latest or an example digest when the registry cannot provide a real, inspectable reference.

28.2 Create the Production Compose Override

Keep one production base file and choose exactly one small HTTPS override. This design makes an accidental combination of local and public host port mappings less likely. The source ZIP includes the inherited application and complete new files; the listings below are the exact additions needed to reproduce this stage without GitHub.

The production base

LISTING • compose.prod28.yaml (140 lines)

```
# Chapter 28 base: no published ports, no build on deployment host.
# Use with EXACTLY ONE of compose.local28.yaml or compose.public28.yaml.
name: deployboard-prod28
x-app-env: &app_env
  DB_HOST: db
  DB_NAME: deployboard
  DB_USER: deployboard
  DB_PASSWORD_FILE: /run/secrets/db_password
  REDIS_HOST: redis
x-app-safety: &app_safety
```

```
  read_only: true
  cap_drop: [ALL]
  security_opt: ["no-new-privileges:true"]
  tmpfs: ["/tmp:rw,nosuid,noexec,size=32m,mode=1777"]
  pids_limit: 128
x-logs: &logs
  driver: local
  options: {max-size: "10m", max-file: "3"}
services:
  db:
    image: ${DEPLOYBOARD_POSTGRES_IMAGE:-postgres:17}
    environment:
      POSTGRES_DB: deployboard
      POSTGRES_USER: deployboard
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets: [db_password]
    volumes: ["dbdata:/var/lib/postgresql/data"]
    networks: [private]
    logging: *logs
    restart: unless-stopped
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
      interval: 5s
      timeout: 3s
      retries: 12
      start_period: 15s
  redis:
    image: ${DEPLOYBOARD_REDIS_IMAGE:-redis:7.4.11-bookworm}
    command: ["redis-server", "--appendonly", "yes", "--appendfsync", "everysec"]
    volumes: ["redisdata:/data"]
    networks: [private]
    logging: *logs
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 12
  api:
    <<: *app_safety
    image: ${DEPLOYBOARD_IMAGE:?Set a published digest-pinned image}
    environment:
      <<: *app_env
      DEPLOYBOARD_VIEWER_TOKEN_FILE: /run/secrets/viewer_token
      DEPLOYBOARD_OPERATOR_TOKEN_FILE: /run/secrets/operator_token
      DEPLOYBOARD_WRITE_LIMIT_PER_MINUTE: "30"
    secrets: [db_password, viewer_token, operator_token]
    depends_on:
      db: {condition: service_healthy}
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8000/ready',timeout=3)"]
      interval: 10s
      timeout: 4s
      retries: 4
      start_period: 20s
    networks: [edge, private]
    logging: *logs
    restart: unless-stopped
    mem_limit: 384m
    cpus: 1.0
  publisher:
```

```
  <<: *app_safety
  image: ${DEPLOYBOARD_IMAGE:?Set a published digest-pinned image}
  command: ["python", "-m", "app.publisher"]
  environment: *app_env
  secrets: [db_password]
  depends_on:
    db: {condition: service_healthy}
    redis: {condition: service_healthy}
  networks: [private]
  logging: *logs
  restart: unless-stopped
  mem_limit: 256m
  cpus: 0.5
 worker:
  <<: *app_safety
  image: ${DEPLOYBOARD_IMAGE:?Set a published digest-pinned image}
  command: ["python", "-m", "app.worker"]
  environment: *app_env
  secrets: [db_password]
  depends_on:
    db: {condition: service_healthy}
    redis: {condition: service_healthy}
  stop_grace_period: 20s
  networks: [private]
  logging: *logs
  restart: unless-stopped
  mem_limit: 256m
  cpus: 0.5
 proxy:
  image: ${DEPLOYBOARD_CADDY_IMAGE:-caddy:2.11.4-alpine}
  volumes:
    - ./site:/srv:ro
    - caddydata:/data
    - caddyconfig:/config
  environment:
    DEPLOYBOARD_DOMAIN: ${DEPLOYBOARD_DOMAIN:-localhost}
  depends_on:
    api: {condition: service_healthy}
  networks: [edge]
  logging: *logs
  restart: unless-stopped
  mem_limit: 192m
  cpus: 0.5
  cap_drop: [ALL]
  cap_add: [NET_BIND_SERVICE]
  security_opt: ["no-new-privileges:true"]
 migrate:
  <<: *app_safety
  profiles: [tools]
  image: ${DEPLOYBOARD_IMAGE:?Set a published digest-pinned image}
  command: ["alembic", "upgrade", "head"]
  environment: *app_env
  secrets: [db_password]
  depends_on:
    db: {condition: service_healthy}
  networks: [private]
  logging: *logs
networks:
 edge: {driver: bridge}
 private: {driver: bridge, internal: true}
volumes:
```

```
  dbdata:
  redisdata:
  caddydata:
  caddyconfig:
secrets:
  db_password: {file: ./private/db_password.txt}
  viewer_token: {file: ./private/viewer_token.txt}
  operator_token: {file: ./private/operator_token.txt}
```

The base has no published ports. All three application roles use the same digest; the migration service uses that digest too, so schema code cannot silently differ from runtime code. PostgreSQL and Redis have named data volumes, but they do not have a host-facing socket.

28.3 Configure HTTPS with a Local CA or Public Certificates

For a free, offline-capable route, expose the proxy only on loopback port 8443. Caddy issues the localhost certificate from its internal CA; its root is not trusted by the host automatically. The Caddy data volume persists the CA identity and certificates.

LISTING • compose.local28.yaml (7 lines)

```
# Free HTTPS via Caddy's local CA; loopback only, never public.
services:
  proxy:
    ports:
      - "127.0.0.1:8443:443"
    volumes:
      - ./proxy/Caddyfile.local28:/etc/caddy/Caddyfile:ro
```

LISTING • proxy/Caddyfile.local28 (16 lines)

```
localhost {
    tls internal
    encode zstd gzip
    log {
        output stdout
        format json
    }
    @api path /health /ready /v1/* /docs /openapi.json
    handle @api {
        reverse_proxy api:8000
    }
    handle {
        root * /srv
        file_server
    }
}
```

Local certificate verification

LISTING • Terminal — local CA check (5 lines)

```
# AFTER the local stack is running:
BASE='docker compose -p deployboard-prod28 -f compose.prod28.yaml -f compose.local28.yaml'
$BASE cp proxy:/data/caddy/pki/authorities/local/root.crt ./local-root-ca.crt
curl --cacert ./local-root-ca.crt https://localhost:8443/ready
# Expected body after migrations: {"status":"ready","storage":"postgresql"}
```

Treat local-root-ca.crt as an explicit trust anchor. Use curl --cacert for the reproducible check; do not normalize curl -k as a “verification” method. Do not publish your Caddy internal CA private key or install an unknown CA root into the system trust store.

28.4 Distinguish Free Local HTTPS from DNS-Based Public TLS

Public HTTPS is an optional, separate exercise. You need a domain that resolves to your server, inbound TCP 80 and 443 reachability for certificate provisioning and redirects, and host firewall/network ACL rules that permit only approved traffic. Caddy automates certificate issuance and renewal; a domain name in a local Caddyfile does not itself provide internet connectivity.

LISTING • compose.public28.yaml (8 lines)

```
# Public TLS only after DNS and firewall approval.
services:
 proxy:
   ports:
     - "80:80"
     - "443:443"
   volumes:
     - ./proxy/Caddyfile.public28:/etc/caddy/Caddyfile:ro
```

LISTING • proxy/Caddyfile.public28 (16 lines)

```
{$DEPLOYBOARD_DOMAIN} {
   encode zstd gzip
   header Strict-Transport-Security "max-age=31536000"
   log {
      output stdout
      format json
   }
   @api path /health /ready /v1/* /docs /openapi.json
   handle @api {
      reverse_proxy api:8000
   }
   handle {
      root * /srv
      file_server
   }
}
```

Choice	TLS issuer	Published host ports	Appropriate proof

Choice	TLS issuer	Published host ports	Appropriate proof
Local VM	Caddy internal CA	127.0.0.1:8443 only	curl --cacert; test from host
Remote public	Public CA via Caddy	80 and 443 after approval	DNS, trusted HTTPS, external scan

Critical boundary Never reuse the public override on a learning VM without reviewing network exposure. Do not publish 5432 or 6379. HSTS is included on the public site, not on the local CA learning endpoint.

28.5 Keep Database and Redis Unpublished

Networks must agree with the threat model: Caddy and the API share the edge network; the API, publisher, worker, database, and Redis share the internal network. The database and queue have no host port mappings. A container in the same authorized internal network can still reach those services; “private” is not an authentication replacement.

Inspect the merged production graph before startup

LISTING • Terminal — private boundary (8 lines)

```
export DEPLOYBOARD_IMAGE='ghcr.io/OWNER/REPO@sha256:ACTUAL_64_HEX_DIGEST'
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml config -q
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml config --services
# Inspect service ports after startup, without printing the full secret config:
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml ps
```

Avoid sharing raw docker compose config output: resolved environment values and paths can disclose information. `config -q` tests syntax without printing the merged document. After startup, confirm that only proxy has published ports.

Offline policy check (runs without Docker)

LISTING • tools/deploy28_validate.py (35 lines)

```
"""Offline structural approval; no Docker commands or credentials opened."""
import re
from pathlib import Path
import yaml

DIGEST = re.compile(r'^(?:ghcr\.io|[a-z0-9.-]+(?::[0-9]+)?)/[a-z0-9._/-]+@sha256:[a-f0-9]{64}$')
DOMAIN = re.compile(r'^(?=.{4,253}$)(?:[a-z0-9](?:[a-z0-9-]{0,61}[a-z0-9])?\.)+[a-z]{2,63}$')

def image_ok(ref):
    return bool(DIGEST.fullmatch(ref or '')) and 'example.invalid' not in ref

def domain_ok(domain):
    return bool(DOMAIN.fullmatch(domain or '')) and not domain.endswith('.example.com')
```

```
def verify(root):
    root = Path(root)
    base = yaml.safe_load((root/'compose.prod28.yaml').read_text())
    local = yaml.safe_load((root/'compose.local28.yaml').read_text())
    public = yaml.safe_load((root/'compose.public28.yaml').read_text())
    errors = []
    s = base['services']
    if base.get('name') != 'deployboard-prod28':
        errors.append('wrong Compose project name')
    for service in ('db', 'redis', 'api', 'publisher', 'worker', 'migrate'):
        if s[service].get('ports') or s[service].get('build'):
            errors.append(service + ' must not publish a port or build on host')
    if s['proxy'].get('ports'):
        errors.append('base proxy publishes ports')
    if not base['networks']['private'].get('internal'):
        errors.append('missing internal database network')
    if local['services']['proxy'].get('ports') != ['127.0.0.1:8443:443']:
        errors.append('local TLS must bind loopback only')
    if public['services']['proxy'].get('ports') != ['80:80', '443:443']:
        errors.append('public TLS should publish only 80 and 443')
    return errors
```

28.6 Launch Database Migrations and Application Services Safely

Compose `depends_on: service_healthy` checks startup health; it is neither a transaction coordinator nor an ongoing failover controller. Because /ready checks PostgreSQL tables, a fresh API might remain unready until the migrations run. Bootstrap database and Redis first, perform one-off Alembic migrations, and only then start API, publisher, worker, and proxy.

Before touching an existing database Take and verify a PostgreSQL backup, note the currently running image digest, review each migration for backward compatibility, and decide how active jobs will be drained or reconciled. PostgreSQL backups do not capture the Redis queue state. Schedule a quiet window if status-changing background jobs are in flight.

Consistent backup from an existing host deployment

LISTING • Terminal — recovery point (14 lines)

```
# Run on the EXISTING deployment and use the same project/file identity.
umask 077
mkdir -p backups
# Example uses the Chapter 28 project; select your real active Compose files.
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml \
  exec -T db sh -c 'PGPASSWORD="$(cat /run/secrets/db_password)" \
    exec pg_dump -h 127.0.0.1 -U deployboard -d deployboard -Fc' \
  > backups/deployboard-before-28.dump
# Hold if the dump is empty or fails its structural inspection.
test -s backups/deployboard-before-28.dump
cat backups/deployboard-before-28.dump | docker compose -p deployboard-prod28 \
```

```
-f compose.prod28.yaml -f compose.local28.yaml \
exec -T db pg_restore -l > /dev/null
```

For an entirely fresh project there is no prior database to back up; verify your storage and restore plan before calling the host production-ready. A backup that has never been restored in a rehearsal is only a candidate recovery point.

The guarded deployment launcher

LISTING • tools/deploy28.py (65 lines)

```
"""Deploy only on an already approved host; never creates secrets or deletes volumes."""
import argparse
import os
from pathlib import Path
import subprocess
import sys
if __package__:
    from .deploy28_validate import image_ok, domain_ok, verify
else:
    from deploy28_validate import image_ok, domain_ok, verify

ROOT = Path(__file__).resolve().parents[1]

def arguments(mode):
    return ['docker', 'compose', '-p', 'deployboard-prod28',
            '-f', 'compose.prod28.yaml', '-f', 'compose.'+mode+'28.yaml']

def plan(mode, image, domain=None):
    errors = verify(ROOT)
    if not image_ok(image):
        errors.append('DEPLOYBOARD_IMAGE must be a real, immutable digest')
    if mode == 'public' and not domain_ok(domain):
        errors.append('public mode requires a real public DNS hostname')
    return errors

def main(argv=None):
    ap = argparse.ArgumentParser()
    ap.add_argument('mode', choices=['local', 'public'])
    ap.add_argument('--image', required=True)
    ap.add_argument('--domain', default=None)
    ap.add_argument('--run', action='store_true', help='execute after review')
    a = ap.parse_args(argv)
    problems = plan(a.mode, a.image, a.domain)
    for name in ('db_password', 'viewer_token', 'operator_token'):
        if not (ROOT/'private'/f'{name}.txt').is_file() and a.run:
            problems.append('missing host-managed private/'+name+'.txt')
    if problems:
        for p in problems: print('HOLD:', p, file=sys.stderr)
        return 2
    cmd = arguments(a.mode)
    env = {**os.environ, 'DEPLOYBOARD_IMAGE': a.image,
           'DEPLOYBOARD_DOMAIN': a.domain or 'localhost'}
    print('Validated configuration; project deployboard-prod28; mode', a.mode)
    print('Source image digest:', a.image)
    if not a.run:
        print('DRY RUN: add --run only after backups, port checks and approval.')
        return 0
    actions = [
```

```
    cmd+['config', '-q'],
    cmd+['--profile','tools','pull'],
    cmd+['up','-d','db','redis'],
    cmd+['--profile','tools','run','--rm','migrate'],
    cmd+['up','-d','api','publisher','worker','proxy'],
    cmd+['ps'],
  ]
  for argv in actions:
     result = subprocess.run(argv, cwd=ROOT, env=env, check=False)
     if result.returncode:
        print('HOLD: command failed; no automatic data deletion or rollback', file=sys.stderr)
        return result.returncode or 1
  print('STARTED, NOT APPROVED: run HTTPS/auth/job smoke tests.')
  return 0

if __name__ == '__main__':
  sys.exit(main())
```

Rehearse, then execute Run `python3 tools/deploy28.py local --image "$DEPLOYBOARD_IMAGE"` first. After host credentials, backup and approval are in place, add `--run`. For public mode supply `--domain app.yourdomain.tld`; the tool does not generate credentials or delete persistent volumes.

28.7 Verify HTTPS, Readiness, and Core User Operations

Checking 200 from the proxy homepage only proves that Caddy can serve a page. Verify a trusted TLS chain and the API readiness route, then prove an authenticated database read and a deliberate write. The independent smoke script verifies certificate trust and returns a nonzero exit status when required checks fail.

LISTING • tools/smoke28.py (61 lines)

```
"""No external dependencies. Host HTTPS/API smoke probe; no tokens printed."""
import argparse
import json
from pathlib import Path
import ssl
import sys
import urllib.error
import urllib.request
import uuid


def call(root, route, token=None, method='GET', payload=None, cafile=None):
  data = json.dumps(payload).encode() if payload is not None else None
  headers = {'Accept': 'application/json', 'X-Request-ID': str(uuid.uuid4())}
  if data is not None: headers['Content-Type'] = 'application/json'
  if token: headers['Authorization'] = 'Bearer '+token
  req = urllib.request.Request(root+route, method=method,
                               data=data, headers=headers)
```

```
    ctx = ssl.create_default_context(cafile=cafile)
    try:
        with urllib.request.urlopen(req, context=ctx, timeout=12) as r:
            raw = r.read()
            return r.status, json.loads(raw) if raw else None, dict(r.headers)
    except urllib.error.HTTPError as e:
        return e.code, None, dict(e.headers)


def main(argv=None):
    p = argparse.ArgumentParser()
    p.add_argument('--url', required=True)
    p.add_argument('--ca-file')
    p.add_argument('--private-dir', default='private')
    p.add_argument('--write-probe', action='store_true',
                   help='creates one uniquely named staging record')
    args = p.parse_args(argv)
    root = args.url.rstrip('/')
    if not root.startswith('https://'):
        print('HOLD: HTTPS URL required');return 2
    folder = Path(args.private_dir)
    def token(name):
        return (folder / (name+'.txt')).read_text().rstrip('\r\n')
    checks = []
    def check(label, actual, expected):
        checks.append(actual == expected)
        print(('PASS' if actual == expected else 'HOLD')+': '+label+
              ' (HTTP '+str(actual)+')')
    try:
        check('ready', call(root,'/ready',cafile=args.ca_file)[0],200)
        check('anonymous blocked',call(root,'/v1/deployments',cafile=args.ca_file)[0],401)
        check('viewer can read',call(root,'/v1/deployments',token('viewer_token'),cafile=args.ca_file)[0],200)
        data={'name':'release-'+uuid.uuid4().hex[:10], 'environment':'staging', 'version':'28.0'}
        check('viewer cannot
write',call(root,'/v1/deployments',token('viewer_token'),method='POST',payload=data,cafile=args.ca_file)[0],403)
        if args.write_probe:
            check('operator can
write',call(root,'/v1/deployments',token('operator_token'),method='POST',payload=data,cafile=args.ca_file)[0],201)
    except (OSError, ValueError, ssl.SSLError) as exc:
        print('HOLD: HTTPS or credential check failed:',type(exc).__name__)
        return 2
    return 0 if all(checks) else 2

if __name__ == '__main__':
    sys.exit(main())
```

Run smoke checks without printing bearer tokens

LISTING • Terminal — local release smoke (4 lines)

```
python3 tools/smoke28.py --url https://localhost:8443 \
  --ca-file ./local-root-ca.crt --private-dir ./private
# To create one unique staging record (side effect), add --write-probe.
# Expected: ready 200, anonymous 401, viewer read 200, viewer write 403.
```

For the public route omit `--ca-file` and set `--url https://app.yourdomain.tld`. The script reads credentials from local host files and does not echo them. Monitor HTTP response codes, request IDs, and application logs without printing headers or tokens.

28.8 Verify Authenticated Access, Denied Requests, and Request-Rate Controls

DeployBoard retains two roles: viewer can read and operator can modify records. Your expected outcomes are 401 with no token, 403 for a viewer attempting POST, and 201 for a valid operator POST. A public service additionally needs edge flood protection; the small code below enforces a basic write limit per token in this single-process demonstration. It is NOT a shared Redis counter, bot defense, or network-level DoS protection.

LISTING • app/limit28.py (31 lines)

```
"""Small single-process mutation limiter; not a distributed WAF or DDoS defense."""
from collections import defaultdict, deque
from hashlib import sha256
from threading import Lock
import time

class WindowLimiter:
    def __init__(self, limit=30, window=60, max_keys=2048):
        if not (1 <= limit <= 10000):
            raise ValueError('limit outside supported range')
        self.limit = limit
        self.window = window
        self.max_keys = max_keys
        self.data = defaultdict(deque)
        self.lock = Lock()

    def allow(self, identity, now=None):
        now = time.monotonic() if now is None else now
        key = sha256(identity.encode('utf8')).digest()
        with self.lock:
            if len(self.data) > self.max_keys:
                for k in list(self.data):
                    if not self.data[k] or self.data[k][-1] <= now - self.window:
                        del self.data[k]
            q = self.data[key]
            while q and q[0] <= now - self.window:
                q.popleft()
            if len(q) >= self.limit:
                return False
            q.append(now)
            return True
```

In app/main.py, add this import and middleware immediately after FastAPI initialization

LISTING • app/main.py — exact inserted patch (16 lines)

```
# Add imports near the other imports:
import os
from fastapi.responses import JSONResponse
from .limit28 import WindowLimiter

# Within create_app, immediately after api = FastAPI(...):
    # A small, explicit single-process mutation limit. Do not treat as edge WAF.
    limit = WindowLimiter(int(os.getenv("DEPLOYBOARD_WRITE_LIMIT_PER_MINUTE", "30")))
    @api.middleware("http")
```

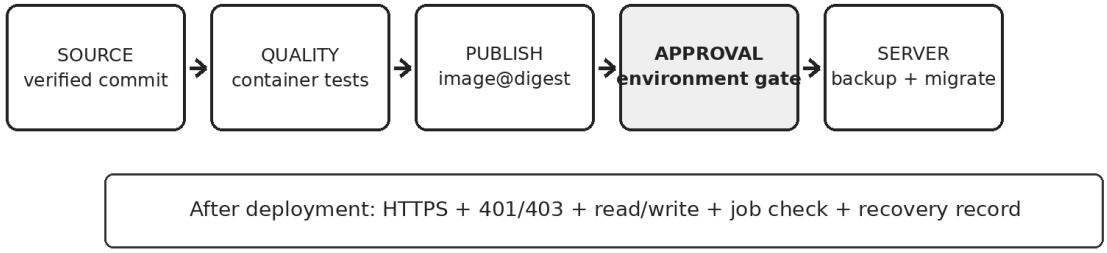
```
async def write_rate_limit(request: Request, call_next):
    if request.url.path.startswith("/v1/") and request.method in ("POST", "PATCH", "DELETE"):
        identity = request.headers.get("authorization", "anonymous")
        if not limit.allow(identity):
            return JSONResponse({"detail": "Write request rate exceeded"}, status_code=429,
                                headers={"Retry-After": "60"})
    return await call_next(request)
```

This limiter counts write requests against a hashed authorization header, not a public IP. It returns 429 and Retry-After when the configured window is exceeded. A single Uvicorn process has one counter; deploying multiple replicas requires a shared rate-limit service or gateway. Test the rate rule with the included unit fixture rather than saturating your production API.

28.9 Add a Gated Deployment Step to the CI/CD Workflow

Add deployment as a dependent job in the existing Chapter 26 workflow. It must not run for pull requests or arbitrary branch pushes. The publisher must have passed the container quality gate and emitted an immutable digest. A protected GitHub environment can require human approval before the SSH deployment step receives environment credentials (subject to repository-plan support).

FIGURE 28.2 | THE RELEASE GATE



CI builds and tests before publication. The deployment job waits for the publication digest, environment policy, and server-side release checks before it may be approved.

Chapter 26 publish job: expose image and digest outputs

LISTING • YAML — add under the existing publish job (7 lines)

```
publish:
  name: Publish approved tag to GHCR
  needs: [gate]
  outputs:
    image: ${{ steps.release.outputs.image }}
    digest: ${{ steps.build.outputs.digest }}
  # Keep all existing steps and tag/ancestry checks from Chapter 26.
```

Add the complete deploy job under jobs: in .github/workflows/deployboard-ci.yml
LISTING • .github/workflows/deployboard-ci.yml — new deploy job (48 lines)

```
deploy:
  name: Gated release to approved Linux host
  needs: [gate, publish]
  if: >-
    ${{ github.event_name == 'push' && startsWith(github.ref, 'refs/tags/v') }}
  runs-on: ubuntu-24.04
  timeout-minutes: 20
  environment: production
  concurrency:
    group: deployboard-production
    cancel-in-progress: false
  permissions:
    contents: read
  steps:
    - name: Validate immutable publish outputs and host inputs
      shell: bash
      env:
        IMAGE_NAME: ${{ needs.publish.outputs.image }}
        DIGEST: ${{ needs.publish.outputs.digest }}
        DEPLOY_HOST: ${{ vars.DEPLOY_HOST }}
        DEPLOY_USER: ${{ vars.DEPLOY_USER }}
        DEPLOY_DOMAIN: ${{ vars.DEPLOY_DOMAIN }}
      run: |
        [[ "$DEPLOY_DOMAIN" =~ ^[a-z0-9.-]+$ ]] || exit 2
        [[ "$IMAGE_NAME" =~ ^ghcr\.io/[a-z0-9._/-]+$ ]] || exit 2
        [[ "$DIGEST" =~ ^sha256:[a-f0-9]{64}$ ]] || exit 2
        [[ "$DEPLOY_HOST" =~ ^[a-zA-Z0-9.-]+$ ]] || exit 2
        [[ "$DEPLOY_USER" =~ ^[a-zA-Z0-9_-]+$ ]] || exit 2
        echo "RELEASE_IMAGE=$IMAGE_NAME@$DIGEST" >> "$GITHUB_ENV"
    - name: Deploy without transferring database credentials
      shell: bash
      env:
        DEPLOY_HOST: ${{ vars.DEPLOY_HOST }}
        DEPLOY_USER: ${{ vars.DEPLOY_USER }}
        DEPLOY_DOMAIN: ${{ vars.DEPLOY_DOMAIN }}
        DEPLOY_KEY: ${{ secrets.DEPLOY_SSH_KEY }}
        KNOWN_HOSTS: ${{ secrets.DEPLOY_KNOWN_HOSTS }}
      run: |
        set -euo pipefail
        umask 077
        mkdir -p "$HOME/.ssh"
        printf '%s\n' "$DEPLOY_KEY" > "$HOME/.ssh/deploy_id"
        printf '%s\n' "$KNOWN_HOSTS" > "$HOME/.ssh/known_hosts"
        ssh -F /dev/null -i "$HOME/.ssh/deploy_id" \
          -o IdentitiesOnly=yes -o StrictHostKeyChecking=yes \
          -o UserKnownHostsFile="$HOME/.ssh/known_hosts" \
          "$DEPLOY_USER@$DEPLOY_HOST" \
          "cd /opt/deployboard && python3 tools/deploy28.py public --image '$RELEASE_IMAGE' --domain
$DEPLOY_DOMAIN --run"
```

The workflow assumes that `/opt/deployboard` is provisioned with reviewed, root-controlled source and host-managed credentials before the job is enabled; the server can pull the private registry image using preconfigured read-only credentials. Do not let an untrusted pull request replace host scripts or write to your deployment directory.

28.10 Protect Deployment Credentials and Environment Approvals

Create a GitHub environment named production. When supported, require an independent reviewer, restrict permitted deployment tags, and deny self-approval. Set DEPLOY_HOST, DEPLOY_USER, DEPLOY_DOMAIN as environment variables and store DEPLOY_SSH_KEY plus an out-of-band verified DEPLOY_KNOWN_HOSTS entry as environment secrets. Register the host SSH fingerprint before any automation; accepting an unverified ssh-keyscan result defeats host authenticity checks.

Identity	Minimum function	Do not grant
CI GITHUB_TOKEN	Read repository / publish only in publish job	Unrestricted write or deploy SSH credentials on PRs
Deploy SSH account	Run a reviewed, tightly scoped deploy action	Interactive administration of all host services
Host registry credential	Read exactly the required private image	Registry owner/admin or global push rights
Host application secrets	Readable only by intended services and trusted host admins	Transfer back to GitHub build artifacts

Privilege reality A Unix account with unrestricted Docker daemon access is effectively root-equivalent. For a true production host, restrict the deploy principal to an approved, root-owned deployment entrypoint or a narrowly scoped deployment agent. The example SSH command assumes the dedicated account has explicitly approved rights; it is not an automatic permission boundary.

Plan-dependent environment rules matter: on some GitHub plans and private repositories, required-reviewer protection may be unavailable. In that case, enforce approval outside GitHub Actions, retain a manual deploy trigger, or upgrade to a plan supporting the rule—do not silently remove the gate.

28.11 Run Post-Deployment Smoke Tests

Run the smoke tests from the actual client network you intend to support. A loopback success on the server says nothing about internet reachability; a public HTTPS result says nothing about database durability or the worker queue. Check four independent paths in order.

TLS and routing Trusted HTTPS certificate; /ready is 200; proxy logs show request ID.

Authorization Anonymous read 401, viewer read 200, viewer write 403; operator write 201 only in the optional probe.

Persistence Recreate only the API container, then re-read the same record ID; leave named volumes untouched.

Asynchronous work Submit a unique check job and observe queued → completed (or a documented, bounded failure) while worker and publisher logs are correlated.

LISTING • Terminal — deploy verification (9 lines)

```
# Run against the selected release project after deployment:
docker compose -p deployboard-prod28 -f compose.prod28.yaml \
  -f compose.local28.yaml ps
docker compose -p deployboard-prod28 -f compose.prod28.yaml \
  -f compose.local28.yaml logs --tail=80 api publisher worker proxy
# Recreate ONLY API after recording a test record ID:
docker compose -p deployboard-prod28 -f compose.prod28.yaml \
  -f compose.local28.yaml up -d --no-deps --force-recreate api
# Repeat smoke28 and retrieve the same record by its ID.
```

Operator checkpoint The endpoint POST /v1/deployments/{id}/checks?key=... accepts an operator token, and GET /v1/checks/{job_id} accepts a viewer token. Observe the job until terminal state; do not pretend `depends_on` provides retries or processing guarantees.

28.12 Record the Deployment Manifest and Recovery Point

Store the running image digest, deployment mode, backup file hash, change approver, and final acceptance results in a release evidence record. A filename by itself is not a backup guarantee; independently verify the file, secure it, and rehearse restoration on a disposable host. Record compatibility limits for rolling back binaries across changed schemas.

LISTING • tools/manifest28.py (45 lines)

```
"""Write a no-secrets release evidence record without inventing runtime proof."""
import argparse
from datetime import datetime, timezone
import hashlib
import json
from pathlib import Path
import sys
if __package__:
    from .deploy28_validate import image_ok
```

```
else:
    from deploy28_validate import image_ok


def make_record(image, mode, backup, previous=''):
    if not image_ok(image) or mode not in ('local','public'):
        raise ValueError('invalid image or deployment mode')
    path = Path(backup)
    if not path.is_file():
        raise ValueError('backup does not exist')
    h = hashlib.sha256()
    with path.open('rb') as f:
        for block in iter(lambda: f.read(1024*1024), b''):
            h.update(block)
    return {'recorded_utc': datetime.now(timezone.utc).isoformat(),
            'image': image, 'mode': mode,
            'backup_name': path.name, 'backup_sha256': h.hexdigest(),
            'previous_image': previous,
            'manual_checks': {'https': 'PENDING', 'auth': 'PENDING',
                              'jobs': 'PENDING', 'restore_rehearsal': 'PENDING'}}

def main():
    p=argparse.ArgumentParser()
    p.add_argument('--image', required=True)
    p.add_argument('--mode', choices=['local','public'],required=True)
    p.add_argument('--backup',required=True)
    p.add_argument('--previous',default='')
    p.add_argument('--out',default='release28_record.json')
    a=p.parse_args()
    try: record=make_record(a.image,a.mode,a.backup,a.previous)
    except ValueError as e:
        print('HOLD:',e,file=sys.stderr);return 2
    Path(a.out).write_text(json.dumps(record, indent=2)+'\n')
    print('Evidence recorded; all manual checks remain PENDING.')
    return 0
if __name__ == '__main__': sys.exit(main())
```

LISTING • Terminal — release record (5 lines)

```
python3 tools/manifest28.py --image "$DEPLOYBOARD_IMAGE" \
  --mode local --backup backups/deployboard-before-28.dump \
  --out release28_record.json
# Edit release28_record.json ONLY after collecting the real check evidence.
# Never put bearer tokens or database passwords in the manifest.
```

If the migration is not backward-compatible, redeploying an older image by digest is not a complete rollback. Use a rehearsed database restore, coordinate with outstanding queue jobs, and explicitly account for any post-backup writes. Do not run irreversible rollbacks automatically from the CI job.

28.13 Checkpoint: Approve the Local or Public Deployment Release

The release is approved only when recorded evidence—not the existence of the Compose containers—supports every relevant acceptance condition.

Gate	Pass condition	Evidence to retain
Image identity	App and migration image use same real registry digest	Manifest index, source commit, review record
Network boundary	DB/Redis have zero host-published ports	docker compose ps + external port check
Transport	Locally trusted CA OR publicly trusted DNS TLS	curl verified URL, issuer, valid name
Storage/migrations	Backup validated, migration successful, original data visible	Dump hash, Alembic revision, record ID
Authentication	401/403 correctly enforced; authorized actions succeed	Sanitized smoke output
Worker	New background job reaches a terminal status	Job ID, correlated worker logs
CI approval	Quality gate + digest publication + independent approval	Run URL, approver, event/tag
Recovery	Restore procedure reviewed and exercised separately	Recovery-point record and rehearsal date

Local test contract (the complete test file is provided in the manuscript)

LISTING • tests/test_deploy28.py (62 lines)

```
"""Offline safety and limited-rate checks; live Docker/TLS proof is separate."""
from pathlib import Path
from unittest.mock import patch
import pytest
from tools.deploy28_validate import verify, image_ok, domain_ok
from tools.deploy28 import arguments, plan
from tools.manifest28 import make_record
from app.limit28 import WindowLimiter

ROOT = Path(__file__).parents[1]
GOOD = 'ghcr.io/example/deployboard@sha256:'+'a'*64

def test_base_and_overrides_preserve_private_services():
    assert verify(ROOT)==[]

def test_digest_requirements():
    assert image_ok(GOOD)
    for value in ('deployboard:latest','ghcr.io/user/repo@sha256:123','ghcr.io/user/repo:latest'):
        assert not image_ok(value)

def test_public_dns_gate():
    assert domain_ok('app.example.org')
    assert not domain_ok('localhost')
    assert not domain_ok('evil;cmd.test')
    assert plan('public',GOOD,'localhost')
    assert not plan('local',GOOD)

def test_compose_project_and_mode_selection():
```

```
    assert arguments('local')[-2:] == ['-f','compose.local28.yaml']
    assert 'deployboard-prod28' in arguments('public')

def test_limiter_declines_then_recovers():
    lim=WindowLimiter(limit=2,window=60)
    assert lim.allow('Bearer sample',now=0)
    assert lim.allow('Bearer sample',now=2)
    assert not lim.allow('Bearer sample',now=3)
    assert lim.allow('Bearer sample',now=61)
    assert lim.allow('Bearer another',now=3)

def test_release_record_keeps_checks_pending(tmp_path):
    backup=tmp_path/'backup.dump'
    backup.write_bytes(b'fictional backup fixture')
    record=make_record(GOOD,'local',backup)
    assert len(record['backup_sha256'])==64
    assert all(x=='PENDING' for x in record['manual_checks'].values())
    assert 'token' not in str(record).lower()


def test_documented_cli_standalone():
    import subprocess, sys
    p=subprocess.run([sys.executable, 'tools/deploy28.py','local','--image',GOOD],cwd=ROOT,capture_output=True,text=True)
    assert p.returncode == 0, p.stderr
    assert 'DRY RUN' in p.stdout

def test_manifest_cli_standalone(tmp_path):
    import subprocess, sys
    backup=tmp_path/'rehearsal.dump'
    backup.write_bytes(b'test fixture')
    out=tmp_path/'record.json'
    p=subprocess.run([sys.executable,'tools/manifest28.py','--image',GOOD,'--mode','local','--backup',str(backup),'--
out',str(out)],cwd=ROOT,capture_output=True,text=True)
    assert p.returncode == 0, p.stderr
    assert out.is_file()
```

Your deliverable An approved release record with no pending critical checks, or a documented HOLD with the failing gate and corrective action. A local HTTPS deployment is a valid completed learning outcome; you do not need a paid VM, DNS zone, or production internet exposure to finish this chapter.

PART VIII

OPERATE, RECOVER, AND SHIP WITH CONFIDENCE

Chapter 29 — Back Up and Restore Containerized Application Data

29.1 Define Recovery Point and Recovery Time Objectives

A backup is not a recovery result. You need two measurable promises: the recovery point objective (RPO) limits how far back the restored data may be; the recovery time objective (RTO) limits how long the application can remain unavailable. Define these before choosing a tool or a schedule. DeployBoard stores deployment records, status events, and the durable job ledger in PostgreSQL. It uses Redis Streams as a delivery mechanism, so a PostgreSQL dump alone cannot reproduce the entire queue state.

Table 29.1 Sample service objectives—not achieved results

Asset / behavior	Illustrative objective	Required proof
PostgreSQL records	RPO ≤ 24 hours	Successful daily backup and archive verification
Service recovery	RTO ≤ 60 minutes	Timed restore, validation, and service restart drill
Background processing	No unreviewed duplicate side effects	Reconcile ledger and Redis delivery before replay
HTTPS and credentials	Restorable, separately protected	Re-provision secrets and verify TLS trust

Treat the numbers as planning examples, not guarantees. If you need an RPO measured in minutes or recovery to an exact timestamp, add PostgreSQL WAL archiving and point-in-time recovery (PITR); the single database dump in this chapter is not PITR. Obtain explicit approval for the chosen objectives and the expense of meeting them.

29.2 Inventory Persistent Volumes and Database Data

Start by inventorying state rather than assuming every Compose service is disposable. In Chapter 28, the production Compose project is deployboard-

prod28. The PostgreSQL data volume is mounted at /var/lib/postgresql/data in the PostgreSQL 17 container. Redis uses /data; Caddy uses /data and /config. Compose project naming matters: volumes from an isolated recovery project are not automatically the production volumes.

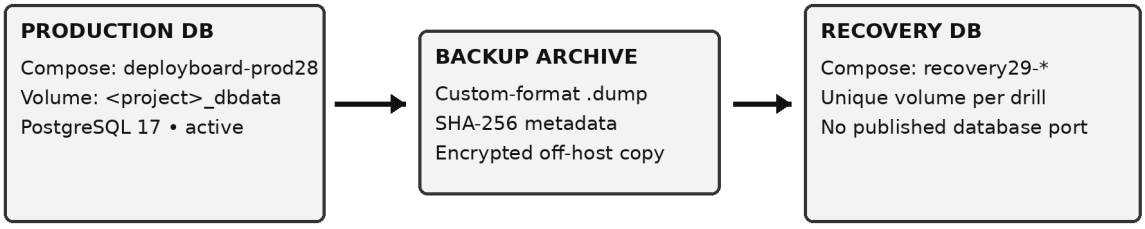
Command 29.1 Inspect running services and named mounts on the Linux host

```
docker compose -p deployboard-prod28 -f compose.prod28.yaml ps
# Run only when the configured project is the intended deployment.
docker volume ls --filter label=com.docker.compose.project=deployboard-prod28
docker inspect "$(docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml ps -q db)" \
  --format '{{range .Mounts}}{{println .Name .Destination}}{{end}}'
```

Table 29.2 What must survive a host failure

Component	Durable state	Recovery approach
PostgreSQL db	deployments, status_events, background_jobs, alembic_version	Logical pg_dump; verified restore
Redis	Stream and pending delivery state; AOF in redisdata	Separate recovery plan or safe queue rehydration
Caddy	TLS certificates, local CA or ACME state	Separate protected backup or authorized reissuance
Secrets and release	File-mounted credentials and image digest	Secure off-host escrow and immutable release record

Figure 29.1 Recovery copies do not share a database volume



Independent recovery project and named-volume isolation.

29.3 Distinguish Database Dumps from File-Level Copies

A live tar archive of PostgreSQL's database directory may capture files at different moments, resulting in an unusable copy. A crash-consistent filesystem snapshot can be valid only when its storage and PostgreSQL recovery assumptions are met; a random hot file copy is not the same as a tested PostgreSQL backup. Use pg_dump for the portable, single-database restore in this chapter. PostgreSQL provides a consistent logical snapshot

while transactions continue, although the snapshot does not include changes committed afterward.

The custom archive (.dump) supports pg_restore, integrity listing, and selectable objects. It does not contain cluster-wide roles or other databases, Redis data, Caddy certificates, Docker images, or host configuration. For a complete disaster-recovery plan, maintain separate inventories and protections for those assets. Do not run schema migrations during the exercise; coordinate write traffic before measuring the recovery point.

COMPATIBILITY BOUNDARY The supplied Compose file uses PostgreSQL 17, so the recovery environment also uses PostgreSQL 17. Review extension, locale, role, and application-version compatibility before restoring to a different PostgreSQL major version.

29.4 Create a Consistent PostgreSQL Backup

Work from the DeployBoard project root on the Linux deployment host. Confirm a previously approved digest-pinned DEPLOYBOARD_IMAGE is exported, the Chapter 28 secrets exist, and the database is running. The script writes PostgreSQL's custom-format archive to a temporary host file with permissions restricted by umask, checks the archive's table of contents, then moves it into place. A command failure removes the incomplete archive.

File 29.1 tools/backup29.sh — complete executable source

```
#!/usr/bin/env bash
# Linux deployment host; from the DeployBoard project root.
set -euo pipefail
umask 077
cd "$(dirname "$0")/.."
: "${DEPLOYBOARD_IMAGE:?Export the deployed digest-pinned image first}"
[[ "$DEPLOYBOARD_IMAGE" == *@sha256:* ]] || { echo 'HOLD: digest image required' >&2; exit 2; }
DEST="${1:-./backups29}"
mkdir -p -- "$DEST"
DEST="$(cd "$DEST" && pwd -P)"
chmod 700 -- "$DEST"
DC=(docker compose -p deployboard-prod28 -f compose.prod28.yaml)
"${DC[@]}" ps -q db | grep -q . || { echo 'HOLD: production db is not running' >&2; exit 2; }
STAMP="$(date -u +%Y%m%dT%H%M%SZ)"
FINAL="$DEST/deployboard-$STAMP.dump"
[[ ! -e "$FINAL" ]] || { echo 'HOLD: name already exists' >&2; exit 2; }
TEMP="$(mktemp "$DEST/.backup29.XXXXXXXX")"
trap 'rm -f -- "$TEMP"' EXIT
# stdout is a binary PostgreSQL archive; never mix log messages into it.
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/db_password)"
  exec pg_dump -h 127.0.0.1 -U "$POSTGRES_USER" \
    -d "$POSTGRES_DB" -Fc -Z 6 --no-owner --no-acl
```

```
' > "$TEMP"
[[ -s "$TEMP" ]] || { echo 'HOLD: empty dump' >&2; exit 2; }
"${DC[@]}" exec -T db pg_restore --list < "$TEMP" > /dev/null
mv -- "$TEMP" "$FINAL"
python3 tools/archive29.py record "$FINAL"
echo "BACKUP RECORDED: $FINAL"
echo 'Transfer securely; a checksum alone is not encryption or proof of trust.'
```

Command 29.2 Run and inspect a database backup

```
export DEPLOYBOARD_IMAGE='YOUR-VERIFIED-IMAGE@sha256:REAL-DIGEST'
bash tools/backup29.sh ./backups29
ls -lh ./backups29/*.dump ./backups29/*.dump.json
```

Expected behavior: one nonempty .dump and a matching .dump.json appear; the script prints the archive location. It must not emit the database password or the dump bytes to terminal output. If Docker is absent, the database is stopped, or the archive check fails, treat the backup attempt as failed. Never turn that error into a success record.

29.5 Protect, Retain, and Transfer Backup Artifacts

A checksum detects changed bytes, but it does not encrypt data or authenticate its original source. The archive can contain sensitive business records. Keep the host backup directory owner-only, exclude all dumps from source control and container build contexts, encrypt copies before moving them off-host, and store decryption material separately. Choose retention by recovery requirement and privacy policy—for example seven daily copies, four weekly copies, and three monthly copies—then test that deletion affects only approved expired archives.

File 29.2 tools/archive29.py — portable SHA-256 manifest and verification

```
"""Create and verify portable integrity metadata; NOT encryption or authenticity."""
from __future__ import annotations
import argparse
from datetime import datetime, timezone
import hashlib
import json
from pathlib import Path
import sys

def digest_file(path: Path) -> str:
    h = hashlib.sha256()
    with path.open('rb') as source:
        for chunk in iter(lambda: source.read(1024 * 1024), b''):
            h.update(chunk)
    return h.hexdigest()


def record(path: Path) -> Path:
    if not path.is_file() or path.stat().st_size < 100:
```

```
        raise ValueError('archive missing or suspiciously small')
    manifest = {
        'format': 'deployboard-pgdump-custom-v1',
        'archive': path.name,
        'bytes': path.stat().st_size,
        'sha256': digest_file(path),
        'created_utc': datetime.now(timezone.utc).isoformat(),
        'postgres_major': 17,
        'scope': 'one PostgreSQL database; Redis and TLS state excluded',
    }
    out = path.with_name(path.name + '.json')
    out.write_text(json.dumps(manifest, indent=2) + '\n', encoding='utf-8')
    out.chmod(0o600)
    return out

def verify(path: Path) -> dict:
    manifest = path.with_name(path.name + '.json')
    data = json.loads(manifest.read_text(encoding='utf-8'))
    if (data.get('format') != 'deployboard-pgdump-custom-v1'
            or data.get('archive') != path.name
            or data.get('bytes') != path.stat().st_size
            or data.get('postgres_major') != 17
            or data.get('sha256') != digest_file(path)):
        raise ValueError('backup metadata mismatch or archive tampering')
    return data

def main(argv=None) -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument('action', choices=['record', 'verify'])
     ap.add_argument('archive', type=Path)
    a = ap.parse_args(argv)
    try:
        result = record(a.archive) if a.action == 'record' else verify(a.archive)
        print('OK:', a.action, a.archive.name)
        return 0
    except (ValueError, OSError, json.JSONDecodeError) as exc:
        print('HOLD:', type(exc).__name__, file=sys.stderr)
        return 2

if __name__ == '__main__':
    sys.exit(main())
```

Command 29.3 Verify the manifest; encrypt the transfer copy (GnuPG)

```
python3 tools/archive29.py verify ./backups29/FILE.dump
# GnuPG prompts for an encryption passphrase; protect it separately.
gpg --symmetric --cipher-algo AES256 \
  --output ./backups29/FILE.dump.gpg ./backups29/FILE.dump
# Transfer the encrypted .gpg and integrity metadata using approved SSH/SFTP.
# Verify the remote copy and document the transfer; do not email plaintext.
```

Never include real archives, recovery passwords, or private keys in this book or the release ZIP. If a backup file must be deleted, do it only after verifying an encrypted off-host copy and following the retention policy. Rotation, automatic cleanup, and off-host access require an accountable operator—not just a cron entry.

29.6 Restore into a Separate Test Environment

Restoration must not require stopping or deleting production. The Compose specification below declares a PostgreSQL-only recovery environment: its own project name (supplied by the restore script), its own database volume, no host-published database port, and a newly generated local test password. Never combine this file with compose.prod28.yaml: that would undermine the isolation boundary.

File 29.3 compose.recovery29.yaml — independent recovery database

```
# PostgreSQL 17 recovery-only database; NEVER combines with prod28 files.
# Each drill uses a unique -p project for fully independent named volumes.
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_USER: deployboard
      POSTGRES_DB: deployboard
      POSTGRES_PASSWORD_FILE: /run/secrets/recovery29_password
    secrets: [recovery29_password]
    volumes: [dbdata:/var/lib/postgresql/data]
    networks: [isolated]
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
      interval: 5s
      timeout: 3s
      retries: 20
        start_period: 10s
networks:
  isolated: {internal: true}
volumes:
  dbdata:
secrets:
  recovery29_password:
    file: ./private/recovery29_db_password.txt
```

File 29.4 tools/restore29.sh — complete isolated restore source

```
#!/usr/bin/env bash
# Restore ONLY into a new, private, independent recovery project.
set -euo pipefail
umask 077
cd "$(dirname "$0")/.."
[[ $# -ge 1 ]] || { echo 'Usage: bash tools/restore29.sh backup.dump [recovery-project]' >&2; exit 2; }
ARCHIVE="$(python3 -c 'from pathlib import Path; import sys; print(Path(sys.argv[1]).resolve())' "$1")"
PROJECT="${2:-deployboard-recovery29-$(date -u +%Y%m%dt%H%M%Sz)}"
[[ "$PROJECT" =~ ^deployboard-recovery29-[a-z0-9-]+$ ]] || { echo 'HOLD: invalid recovery project name' >&2; exit 2; }
python3 tools/archive29.py verify "$ARCHIVE"
DC=(docker compose -p "$PROJECT" -f compose.recovery29.yaml)
if docker volume inspect "${PROJECT}_dbdata" >/dev/null 2>&1; then
  echo 'HOLD: destination volume exists; choose a fresh project' >&2
  exit 2
fi
mkdir -p private
chmod 700 private
PASSFILE=private/recovery29_db_password.txt
if [[ ! -f "$PASSFILE" ]]; then
  python3 - "$PASSFILE" <<'PYCODE'
from pathlib import Path
import secrets, sys
p = Path(sys.argv[1]); p.write_text(secrets.token_urlsafe(36) + '\n')
```

```
p.chmod(0o600)
PYCODE
fi
[[ -s "$PASSFILE" ]] || { echo 'HOLD: empty recovery password' >&2; exit 2; }
"${DC[@]}" up -d db
READY=0
for _ in {1..30}; do
  if "${DC[@]}" exec -T db pg_isready -U deployboard -d deployboard >/dev/null 2>&1; then
    READY=1; break
  fi
  sleep 2
done
[[ "$READY" -eq 1 ]] || { echo 'HOLD: recovery database not ready' >&2; exit 2; }
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/recovery29_password)"
  exec pg_restore -h 127.0.0.1 -U "$POSTGRES_USER" \
    -d "$POSTGRES_DB" --no-owner --no-acl \
    --exit-on-error --single-transaction
' < "$ARCHIVE"
echo "RECOVERY RESTORED: $PROJECT (not production)"
bash tools/verify29.sh "$PROJECT"
echo "NEXT: log measured recovery time and preserve this project for inspection."
```

Give the restore tool a verified archive from your actual backups directory. It creates a new project with a timestamped name, refuses pre-existing recovery volumes, starts an isolated database, waits for readiness, and feeds the archive to pg_restore with fail-fast and single-transaction behavior. The tool never invokes down --volumes on the live stack.

Command 29.4 Run a rehearsal from a stored archive

```
bash tools/restore29.sh ./backups29/FILE.dump
# Record the printed recovery project name for later checks.
# Do not substitute the production Compose project name.
```

SECURITY REVIEW PostgreSQL warns that restoring SQL from an untrusted dump can execute commands supplied by the source. Restore only trusted, controlled backup archives, with access to a disposable environment and an explicit review gate.

29.7 Verify Record Integrity After Restoration

A successful pg_restore exit code only proves that the restore operation completed without an error reported to the client. It does not prove correct business state. Verify the schema revision, row counts, foreign-key relationships, and at least two identifiable application records recorded before backup. Choose records that include a status-event history and, where present, a completed background job. Compare them to a separately preserved, redacted snapshot taken near the recovery point.

File 29.5 tools/verify29.sh — complete restored-data diagnostic

```
#!/usr/bin/env bash
set -euo pipefail
```

```
cd "$(dirname "$0")/.."
[[ $# -eq 1 ]] || { echo 'Usage: bash tools/verify29.sh recovery-project' >&2; exit 2; }
PROJECT="$1"
[[ "$PROJECT" =~ ^deployboard-recovery29-[a-z0-9-]+$ ]] || { echo 'HOLD: invalid recovery project' >&2; exit 2; }
DC=(docker compose -p "$PROJECT" -f compose.recovery29.yaml)
# SQL contains no user-supplied identifiers; no database port is published.
SQL="SELECT 'deployments', count(*) FROM deployments
UNION ALL SELECT 'status_events', count(*) FROM status_events
UNION ALL SELECT 'background_jobs', count(*) FROM background_jobs
UNION ALL SELECT 'missing_event_parent', count(*) FROM status_events e
  LEFT JOIN deployments d ON d.id=e.deployment_id WHERE d.id IS NULL
UNION ALL SELECT 'missing_job_parent', count(*) FROM background_jobs j
  LEFT JOIN deployments d ON d.id=j.deployment_id WHERE d.id IS NULL;"
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/recovery29_password)"
   exec psql -X -v ON_ERROR_STOP=1 -h 127.0.0.1 \
    -U "$POSTGRES_USER" -d "$POSTGRES_DB" -At -F "|" -c "$1"
' sh "$SQL"
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/recovery29_password)"
  exec psql -X -v ON_ERROR_STOP=1 -h 127.0.0.1 \
    -U "$POSTGRES_USER" -d "$POSTGRES_DB" -At \
    -c "SELECT version_num FROM alembic_version"
'
echo 'CHECK: orphan counts must be zero; compare sample IDs and records with release evidence.'
```

Command 29.5 Read restored data without publishing the database port

```
bash tools/verify29.sh deployboard-recovery29-REPLACE-WITH-ACTUAL-NAME
# Expected keys: deployments, status_events, background_jobs,
# missing_event_parent=0, missing_job_parent=0; Alembic revision printed.
```

Any missing parent count above zero is a HOLD. Also hold when the recovered Alembic revision is incompatible with the chosen DeployBoard image. Row counts may legitimately differ from live production because the archive represents an earlier point; that difference must fit the approved RPO, not be hidden by comparing against a moving target.

29.8 Plan Backup Schedules and Access Permissions

Use an operator-owned scheduler with a deliberately limited execution identity and an approved Docker access method. Docker group membership and Docker socket access are effectively host-level administrative powers; simply adding a backup user to that group is not a least-privilege solution. An authorized privileged service account or narrowly supervised host task may be more appropriate on a small single host. Restrict access to output folders, the unit definition, backup encryption keys, and off-host storage.

Example 29.1 Systemd timer design (illustrative; adjust paths and identity)

```
# /etc/systemd/system/deployboard-backup29.service
[Unit]
Description=DeployBoard verified PostgreSQL backup
```

```
[Service]
Type=oneshot
WorkingDirectory=/opt/deployboard
EnvironmentFile=/etc/deployboard/release-image.env
ExecStart=/usr/bin/bash /opt/deployboard/tools/backup29.sh /var/backups/deployboard
# Grant Docker access only through an approved host privilege model.

# /etc/systemd/system/deployboard-backup29.timer
[Unit]
Description=Daily DeployBoard backup schedule
[Timer]
OnCalendar=*-*-* 02:30:00
Persistent=true
[Install]
WantedBy=timers.target
```

An enabled timer is not proof that a usable backup exists: monitor exit codes, last success time, artifact age, backup size, off-host arrival, and periodic restore results. The service example assumes an administrator has arranged secure credentials, appropriate unit execution privileges, directory ownership, and a release-image environment file containing no passwords. Do not enable it unchanged.

29.9 Understand Volume Snapshots and Consistency Limits

Volume snapshots may restore large datasets faster than logical dumps when the underlying filesystem or storage platform provides genuine, tested snapshot semantics. Their limitations include matching PostgreSQL and filesystem versions, crash recovery, snapshot timing, write ordering across multiple volumes, and incomplete coverage of files stored outside the snapshot. A tar of a running database directory is not a substitute for coordinated snapshot tooling.

For DeployBoard, PostgreSQL logical backups remain the portable baseline. Redis AOF and Caddy’s certificate data require separate lifecycle decisions. In particular, do not replay queued tasks blindly after database recovery; compare PostgreSQL’s background_jobs ledger against actual external effects and delivery state to prevent duplicates. Image versions, application migrations, environment settings, DNS, secret retrieval and encryption-key escrow also belong in your broader recovery runbook.

29.10 Run a Complete Recovery Drill

Figure 29.2 A passing checksum is only one recovery gate



Sequential evidence gates for an operational recovery exercise.

Conduct the drill as a measured operation: identify the last approved production release; capture a known deployment record and status event; freeze schema changes; back up; copy the encrypted archive to a separate location; verify integrity after transfer; restore into a new isolated project; compare rows and Alembic revision; then verify the compatible API using a separately authorized test deployment, if you need to prove end-to-end business behavior.

Command 29.6 Offline tool tests and real-run evidence sequence

```
python3 -m unittest tests.test_archive29 -v
export DEPLOYBOARD_IMAGE='YOUR-APPROVED-IMAGE@sha256:REAL-DIGEST'
bash tools/backup29.sh ./backups29
python3 tools/archive29.py verify ./backups29/FILE.dump
bash tools/restore29.sh ./backups29/FILE.dump
# Follow up with tools/verify29.sh using the printed project name.
```

Time the sequence yourself and record the result. With one snapshot per day, the observed recovered data age can approach 24 hours. With a slow off-host download or a new host install, RTO may exceed your initial estimate. Repeat after schema changes, host changes, and major release upgrades. If you later remove a rehearsal, first confirm its exact project name and that it owns only recovery volumes; never use a broad Docker prune or production volume deletion.

29.11 Checkpoint: Prove Recoverability with Evidence

Table 29.3 Release gate for recoverability

Gate	Passing evidence	Stop condition
Archive generated	Nonempty custom-format dump; pg_restore list succeeds	Incomplete file or failed dump
Artifact protected	Checksum verified; encrypted off-host copy accounted for	Only local plaintext copy exists
Isolation	New recovery29-* project; no shared volume	Any production volume attached
Data integrity	Expected IDs, counts, links, migration version	Missing records, orphan rows, incompatible schema
Timeliness	Measured age and duration within approved RPO/RTO	Unknown or exceeded objectives
Dependent systems	Redis/TLS/secret recovery gaps explicitly assigned	Unreviewed replay or missing access

Complete recovery29_evidence_template.md with timestamps, archive digest, restored row counts, identity of checked records, the measured duration, off-host transfer proof, and any open Redis or certificate work. Never fill in PASS merely because the script exited successfully. You have a recovery capability when an independent person can reproduce the drill from a protected copy and independently verify the recovered application data.

OPERATIONAL HANDOFF Keep a verified, encrypted backup away from the Docker host; retest it regularly; and do not mistake a backup file for an executable recovery plan. Chapter 30 uses this evidence to design safe rollouts, rollback decisions, and controlled release recovery.

Chapter 30 — Update Applications and Roll Back Failed Releases

PROJECT OUTCOME Safely stage, promote, verify, and recover a digest-pinned DeployBoard release without treating an image rollback as a data restore.

ACCEPTANCE PROOF The isolated fault drill preserves staging records, the previous image remains available, and production decisions require documented recovery and compatibility evidence.

30.1 Separate Application Releases from Data Migrations

An application rollback and a database restore solve different problems. Replacing the API container can reverse a faulty executable or image configuration; it cannot undo PostgreSQL schema changes, deleted rows, or messages already acknowledged by a worker. DeployBoard now has a digest-pinned API, PostgreSQL records, Redis Streams, Caddy HTTPS, and a tested logical-backup path. Your release must coordinate all of them without confusing their recovery clocks.

Table 30.1 — Two release tracks, two recovery decisions

Track	Changed artifact	Recovery constraint
Application	API, publisher and worker container image	May revert to an earlier digest only if schema and contracts remain compatible
Database	Alembic revision and stored records	Often forward-only; recovery uses an explicit data plan
Delivery state	Redis Stream/pending messages and job ledger	Replaying demands idempotency and reconciliation
Edge	Caddy configuration and certificate state	Change separately and confirm TLS continuity

RELEASE CONTRACT No “GO” without a verified previous digest, a newly verified backup, a completed isolated recovery rehearsal, schema compatibility, staging acceptance, and a named rollback owner.

30.2 Choose Versioned Images and Recorded Digests

The release tag is for humans; the registry digest identifies the image bytes (or platform index) you deploy. Keep both the previous approved digest and the candidate digest in a dated record. Do not use latest or rebuild the old tag in place and assume it identifies the earlier artifact. The base Compose file already uses DEPLOYBOARD_IMAGE for the API, publisher, worker and migration job.

Command 30.1 — Confirm the effective images without disclosing secrets

```
export PREVIOUS_IMAGE='REGISTRY/OWNER/deployboard@sha256:PREVIOUS_64_HEX'
export
CANDIDATE_IMAGE='REGISTRY/OWNER/deployboard@sha256:CANDIDATE_64_HEX'
# Substitute two real, independently approved full digests first.
export DEPLOYBOARD_IMAGE="$CANDIDATE_IMAGE"
docker image inspect "$DEPLOYBOARD_IMAGE" --format '{{.Id}}'
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml \
  config --images
# config --images lists the service image references, not release approval.
```

REGISTRY DIGEST VERSUS IMAGE ID: The locally displayed image ID can differ from the registry manifest digest or multi-platform index digest. Preserve the canonical registry reference with @sha256:... and record the actual platform. A successful pull proves availability, not authenticity or application health.

30.3 Stage an Update Against Representative Test Data

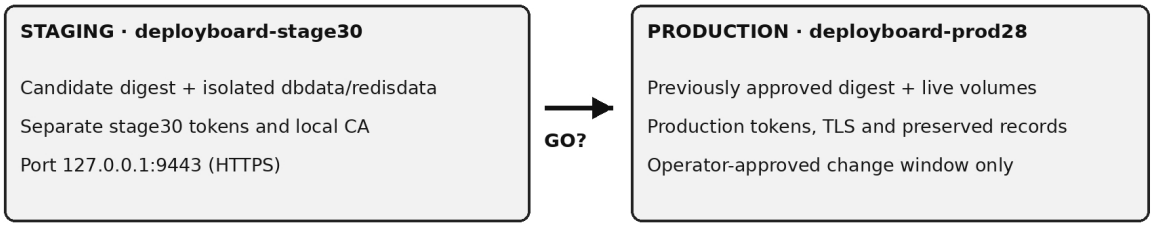
Run the trial on an isolated Compose project so an intentional failure cannot touch the live PostgreSQL volume. The new override does not publish PostgreSQL or Redis; it exposes only Caddy at 127.0.0.1:9443 and points secrets to private/stage30 rather than the production files. The baseline image and application features remain unchanged until the candidate digest is supplied.

File 30.1 — compose.stage30.yaml: complete staging override

```
# Always use -p deployboard-stage30 to isolate volumes and networks.
# Stage uses separate private/stage30/ secret files, not production tokens.
services:
```

```
  proxy:
    ports:
      - "127.0.0.1:${DEPLOYBOARD_STAGE_HTTPS_PORT:-9443}:443"
    volumes:
      - ./proxy/Caddyfile.local28:/etc/caddy/Caddyfile:ro
  api:
    environment:
      DEPLOYBOARD_WRITE_LIMIT_PER_MINUTE: "30"
secrets:
  db_password:
    file: ./private/stage30/db_password.txt
  viewer_token:
    file: ./private/stage30/viewer_token.txt
  operator_token:
    file: ./private/stage30/operator_token.txt
```

Figure 30.1 One release, two independent data environments



A successful stage test does not promote staging data into production.

Command 30.2 — Prepare unique staging secrets and an isolated stack

```
mkdir -p private/stage30
chmod 700 private/stage30
python3 - <<'PYCODE'
from pathlib import Path
import secrets
base = Path('private/stage30')
for name in ('db_password', 'viewer_token', 'operator_token'):
    target = base / (name + '.txt')
    if target.exists():
        raise SystemExit('HOLD: stage secret already exists: ' + name)
    target.write_text(secrets.token_urlsafe(36) + '\n')
    target.chmod(0o600)
PYCODE
export DEPLOYBOARD_IMAGE="$CANDIDATE_IMAGE"
# Keep every Compose invocation explicit; no shell eval.
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml config -q
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml up -d db redis
```

```
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml \
  --profile tools run --rm migrate
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml \
  up -d api publisher worker proxy
```

Expected: a distinct project, separate named volumes, healthy database and API, and a local HTTPS endpoint. Before promoting anything, seed representative *non-sensitive* deployment histories via the Chapter 14 sample seed tool and the Chapter 15 job workflow. Compare staged schema revision and worker behavior with the records captured in your last recovery rehearsal. Synthetic fixtures reveal common regressions; only an authorized, anonymized or safely restored dataset can approximate a complicated production distribution.

Command 30.3 — Stage HTTPS and authenticated acceptance

```
# Export the staging CA from its own Caddy data volume.
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml \
  cp proxy:/data/caddy/pki/authorities/local/root.crt ./stage30-ca.crt
python3 tools/release30_probe.py --url https://localhost:9443 \
  --ca-file ./stage30-ca.crt --private-dir private/stage30 \
  --staging-write
# Inspect a real background job and ensure its terminal status is correct.
# The stage-only write probe creates one staging deployment record.
```

30.4 Plan Forward-Compatible Database Changes

Treat schema changes as an expand → migrate/backfill → contract sequence. Expand first: add nullable fields or new tables while the old API still functions. Deploy the new image only after compatibility checks. Backfill separately, measure progress, and postpone destructive cleanup until the rollback window has closed and no older application requires the field. Avoid assuming an Alembic downgrade will restore deleted or rewritten user data.

Example 30.1 — Illustrative Alembic expand migration (NOT applied by this lab)

```
# Use a NEW Alembic revision in a future application release.
from alembic import op
import sqlalchemy as sa
```

```
def upgrade():
    op.add_column('deployments',
                  sa.Column('release_channel', sa.String(24), nullable=True))

def downgrade():
    # Only safe after confirming this new field has no data to preserve.
    op.drop_column('deployments', 'release_channel')
```

Compatibility proof must cover old app against expanded schema, new app against expanded schema, the current Alembic revision, and the behavior of in-flight background jobs. In the fault drill below, no schema migration is introduced; this cleanly isolates an application/configuration failure from a data migration.

30.5 Deploy an Updated Application Deliberately

Before production changes, re-verify the Chapter 29 backup with its checksum manifest and record the tested restore, establish an operator-owned change window, and obtain approval. The production stack is the fixed project deployboard-prod28, not the staging project. For local HTTPS, use compose.local28.yaml; for an approved public host, substitute compose.public28.yaml. Never merge both TLS overrides.

Command 30.4 — Plan a release; no container mutations

```
python3 tools/release30.py plan \
  --previous "$PREVIOUS_IMAGE" --candidate "$CANDIDATE_IMAGE" \
  --backup ./backups29/VERIFIED.dump --mode local \
  --out release30_plan.json
python3 tools/archive29.py verify ./backups29/VERIFIED.dump
python3 -m unittest tests.test_release30 -v
# Inspect release30_plan.json; it begins in HOLD state.
```

File 30.2 — tools/release30.py: complete, non-deploying release ledger

```
"""Plan, audit, and record releases. This tool never changes running containers."""
import argparse
from datetime import datetime, timezone
import hashlib
import json
from pathlib import Path
import re
import sys

IMAGE = re.compile(r"^[a-z0-9.-]+(?::[0-9]+)?(?:/[a-z0-9._-]+)+@sha256:[a-f0-9]{64}$")
VALID = {"GO", "HOLD", "ROLLBACK", "RECOVER"}

def is_digest(ref):
```

```
    return bool(IMAGE.fullmatch(ref or "")) and "example.invalid" not in ref

def file_sha256(path):
    h = hashlib.sha256()
    with Path(path).open("rb") as handle:
        for block in iter(lambda: handle.read(1024 * 1024), b""):
            h.update(block)
    return h.hexdigest()

def create_plan(previous, candidate, backup, mode):
    if not is_digest(previous) or not is_digest(candidate):
        raise ValueError("both images must use complete sha256 digests")
    if previous == candidate:
        raise ValueError("candidate and previous image must differ")
    archive = Path(backup)
    if not archive.is_file() or archive.stat().st_size == 0:
        raise ValueError("backup archive must exist and be nonempty")
    if mode not in ("local", "public"):
        raise ValueError("mode must be local or public")
    return {
        "recorded_utc": datetime.now(timezone.utc).isoformat(),
        "previous_image": previous,
        "candidate_image": candidate,
        "backup_file": archive.name,
        "backup_sha256": file_sha256(archive),
        "mode": mode,
        "state": "HOLD",
        "evidence": {
            "archive_verified": "PENDING",
            "isolated_restore": "PENDING",
            "schema_compatibility": "PENDING",
            "staging_smoke": "PENDING",
            "post_deploy_smoke": "PENDING",
            "rollback_compatibility": "PENDING"
        },
        "decision_notes": "Not yet approved"
    }

def decide(record, verdict, reason):
    if verdict not in VALID:
        raise ValueError("unknown release decision")
    if not reason.strip():
        raise ValueError("decision requires a reason")
    out = dict(record)
    out["state"] = verdict
    out["decision_notes"] = reason.strip()
    out["decision_utc"] = datetime.now(timezone.utc).isoformat()
    if verdict == "GO" and any(v != "PASS" for v in record["evidence"].values()):
        raise ValueError("GO requires every evidence field to be PASS")
```

```
    return out

def main(argv=None):
    p = argparse.ArgumentParser(description=__doc__)
    sub = p.add_subparsers(dest="cmd", required=True)
    a = sub.add_parser("plan")
    a.add_argument("--previous", required=True)
    a.add_argument("--candidate", required=True)
    a.add_argument("--backup", required=True)
    a.add_argument("--mode", choices=("local", "public"), required=True)
    a.add_argument("--out", default="release30_plan.json")
    b = sub.add_parser("decide")
    b.add_argument("--plan", required=True)
    b.add_argument("--decision", choices=sorted(VALID), required=True)
    b.add_argument("--reason", required=True)
    b.add_argument("--out", default="release30_decision.json")
    args = p.parse_args(argv)
    try:
        if args.cmd == "plan":
            record = create_plan(args.previous, args.candidate, args.backup, args.mode)
        else:
            record = decide(json.loads(Path(args.plan).read_text()),
                            args.decision, args.reason)
        path = Path(args.out)
        if path.exists():
            raise ValueError("output already exists; preserve previous release evidence")
        path.write_text(json.dumps(record, indent=2) + "\n")
        print("RELEASE EVIDENCE WRITTEN:", path, "state:", record["state"])
        return 0
    except (ValueError, OSError, KeyError, json.JSONDecodeError) as exc:
        print("HOLD:", str(exc), file=sys.stderr)
        return 2

if __name__ == "__main__":
    sys.exit(main())
```

Only after all evidence fields are manually reviewed and marked PASS, issue the approved GO decision. This tool refuses to mark a plan GO if any field is still PENDING; it does not turn observations into PASS automatically. Preserve the original plan and decision record unchanged. Do not add a secret, access token, host private key or database password to either JSON file.

Command 30.5 — The controlled production change (approved operators only)

```
# The following is NOT the staging fault injection.
export DEPLOYBOARD_IMAGE="$CANDIDATE_IMAGE"
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml config -q
```

```
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml pull api publisher worker
# Run migrate only if the reviewed candidate migration is explicitly
# forward-compatible and approved (see Section 30.4).
# docker compose -p deployboard-prod28 -f compose.prod28.yaml \
#   -f compose.local28.yaml --profile tools run --rm migrate
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml \
  up -d --no-deps api publisher worker
# Compose replaces selected containers; this is NOT zero-downtime by default.
```

30.6 Run Post-Deployment Health and Functional Checks

A green container health check only proves that its command succeeded at that moment. The acceptance gate also needs an HTTPS certificate check, API readiness, anonymous denial, viewer read, viewer write rejection, operator writes if expressly authorized, and one full queue/job lifecycle. Compare response codes and known record IDs against the pre-deployment snapshot; record actual execution time and outcomes.

File 30.3 — tools/release30_probe.py: complete HTTPS and permission probe

```
"""TLS/auth smoke checks; a write probe is explicitly opt-in for STAGING."""
import argparse
from pathlib import Path
import ssl
import sys
from uuid import uuid4
if __package__:
    from .smoke28 import call
else:
    from smoke28 import call

def run(base, ca, secret_dir, staging_write=False):
    if not base.startswith("https://"):
        raise ValueError("HTTPS URL required")
    root = Path(secret_dir)
    def token(name):
        content = (root / (name + ".txt")).read_text().strip()
        if not content: raise ValueError("missing token")
        return content
    viewer, operator = token("viewer_token"), token("operator_token")
    sample = {"name": "release30-" + uuid4().hex[:10],
              "environment": "staging", "version": "30.0"}
    checks = [
```

```
        ("readiness", "/ready", None, "GET", None, 200),
        ("anonymous denied", "/v1/deployments", None, "GET", None, 401),
        ("viewer read", "/v1/deployments", viewer, "GET", None, 200),
        ("viewer write denied", "/v1/deployments", viewer, "POST", sample, 403),
    ]
    if staging_write:
        checks.append(("staging operator write", "/v1/deployments",
                       operator, "POST", sample, 201))
    ok = True
    for label, path, auth, method, payload, expected in checks:
        result, _, _ = call(base.rstrip("/"), path, auth, method, payload, ca)
        good = result == expected
        ok &= good
        print(("PASS" if good else "HOLD") + ": " + label +
              " (HTTP " + str(result) + ")")
    return 0 if ok else 2

def main(argv=None):
    p = argparse.ArgumentParser()
    p.add_argument("--url", required=True)
    p.add_argument("--ca-file")
    p.add_argument("--private-dir", default="private")
    p.add_argument("--staging-write", action="store_true")
    a = p.parse_args(argv)
    try:
        return run(a.url, a.ca_file, a.private_dir, a.staging_write)
    except (OSError, ValueError, ssl.SSLError) as exc:
        print("HOLD: smoke check failed (" + type(exc).__name__ + ")", file=sys.stderr)
        return 2

if __name__ == "__main__":
    sys.exit(main())
```

Command 30.6 — Observe production without creating new records

```
# For public DNS, specify the official trusted hostname. For the local
# Caddy CA, use --ca-file and the approved local CA certificate.
python3 tools/release30_probe.py --url https://localhost:8443 \
  --ca-file ./approved-prod-ca.crt --private-dir private
# Omit --staging-write in production. Capture exit status and logs.
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml ps
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml \
  logs --tail 80 api publisher worker proxy
```

30.7 Recognize When a Rollback Is Necessary

A rollback decision belongs to the incident owner, using evidence rather than discomfort. Roll back when the candidate repeatedly fails readiness, breaches an approved error/latency threshold, breaks authentication, returns incorrect records, or prevents background jobs from completing. Pause rollout immediately if migration compatibility is unknown, and stop writes if continued operation may corrupt state. If the previous image cannot read the evolved schema, use the recovery plan—not an untested old binary.

Table 30.2 — Clear recovery triggers

Observed evidence	Preferred action	Why
Stage API crash / unhealthy	Restore staging override or known image	Fault is isolated; no production data touched
Production API fails; schema compatible	Recreate from previous approved digest	Code path can be reverted
Destructive schema or lost records	Incident + vetted restore procedure	Old image cannot reverse database state
Redis backlog without DB loss	Stabilize worker; reconcile idempotent jobs	Reverting image alone may replay deliveries

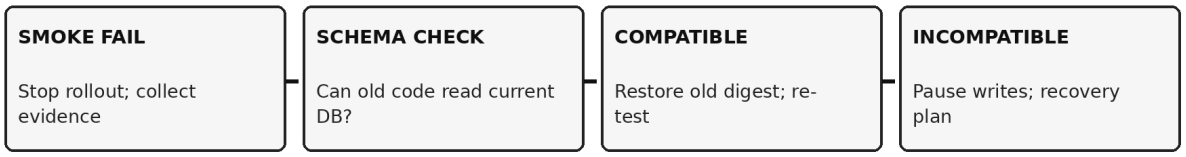
30.8 Restore a Prior Image without Assuming Data Reversibility

The preferred response to a code-only regression is redeploying the previously approved registry digest while preserving PostgreSQL volumes and Redis delivery state. Do not use docker compose restart to apply a changed image reference; Compose restart retains the old container configuration. Do not run down --volumes, and do not downgrade the schema merely because an old image was selected.

Command 30.7 — Application-only rollback after compatibility approval

```
# First: verify old code supports the CURRENT database schema.
# Stop traffic/writes if your approved incident procedure requires it.
export DEPLOYBOARD_IMAGE="$PREVIOUS_IMAGE"
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml \
  pull api publisher worker
docker compose -p deployboard-prod28 \
  -f compose.prod28.yaml -f compose.local28.yaml \
  up -d --no-deps api publisher worker
# Run Section 30.6 probes again and reconcile interrupted jobs.
# Never delete dbdata, redisdata, or Caddy certificate volumes.
```

Figure 30.2 Failed release: separate code recovery from data recovery



Never expect a previous image to reverse ALTER TABLE, destructive migrations, or lost writes.

Rollback is a code decision only after compatibility has been established.

30.9 Use Backup Restoration Only with a Recovery Plan

Use data restoration when live data is damaged or an incompatible schema change makes an application-only rollback unsafe. A logical PostgreSQL dump restores a historical database snapshot: any writes committed after the snapshot may be absent. Obtain RPO/RTO approval, stop or fence writes, preserve the damaged database for investigation, and measure actual recovery time. Restore the archive into a separate Chapter 29 recovery project first, then validate row counts, foreign keys, identifiable records and application compatibility before considering cutover.

Command 30.8 — Safe rehearsal, NOT a production replacement

```
python3 tools/archive29.py verify ./backups29/VERIFIED.dump
bash tools/restore29.sh ./backups29/VERIFIED.dump
# Copy the printed deployboard-recovery29-* project name.
bash tools/verify29.sh deployboard-recovery29-REPLACE
# A good dump does not restore Redis Streams, Caddy keys or new writes.
# Actual production cutover requires a separate reviewed runbook.
```

DATABASE BACKUP IS NOT A TIME MACHINE: Chapter 29 uses a consistent PostgreSQL logical archive, not continuous WAL-based point-in-time recovery. Never promise recovery to the moment before a failed deployment unless the backup strategy actually supports it.

30.10 Document Change Windows and Release Decisions

Use a signed-off release ledger that distinguishes planned actions from observed facts. Record UTC time, actor, previous/candidate images, backup

hash and verified restore, schema revision, traffic impact, exact post-deployment checks, decision, and customer-facing communications. Release metadata is operational evidence, not a place for credentials. The blank form below belongs in the book and the source package.

File 30.4 — release30_change_window.md: complete evidence worksheet

```
# DeployBoard release and rollback decision record

- Change ID / owner / UTC window:
- Previous image digest:
- Candidate image digest:
- Verified backup name and SHA-256:
- Restore drill evidence / RPO / RTO:
- Previous and candidate Alembic revisions:
- Expand / backfill / contract compatibility proof:
- Staging read/write and worker acceptance:
- Go / hold decision and approver:
- Actual deployment start and end UTC:
- HTTPS/ready/auth/read-write/job evidence:
- Rollback trigger and compatibility check:
- Rollback image / action / evidence:
- Remaining data risk and communications:
- Decision: GO / HOLD / ROLLBACK / RECOVER
```

Command 30.9 — Record a decision without overwriting the plan

```
# Edit each evidence field to PASS only after completing its real check.
python3 tools/release30.py decide --plan release30_plan.json \
  --decision HOLD --reason 'Staging acceptance pending' \
  --out release30_hold.json
# An approved GO is accepted only if every evidence item is PASS.
# Preserve decision records; the utility will not overwrite an existing file.
```

30.11 Checkpoint: Recover from an Intentionally Failed Release

Now simulate a failed release safely without publishing a broken registry image. You will run the verified candidate in staging, confirm normal authenticated reads and writes, inject a bad API startup command, capture the expected exit code 42 and degraded readiness, remove the bad override, then recreate only the API. The database, worker, and Redis volumes remain in the staging project throughout. There is no production fault injection.

File 30.5 — compose.fault30.yaml: deliberate STAGING-ONLY startup fault

```
# STAGING-ONLY experiment: force API startup to fail with exit 42.
# NEVER combine this override with -p deployboard-prod28.
services:
  api:
    command: ["python", "-c", "import sys; sys.exit(42)"]
```

Command 30.10 — Break only staging API, inspect, recover

```
export DEPLOYBOARD_IMAGE="$CANDIDATE_IMAGE"
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml \
  -f compose.fault30.yaml up -d --no-deps api
# Confirm crash/restarts or failed health. Do not tear down the project.
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml ps api
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml logs --tail 35 api
# Restore normal command by OMITTING the fault override.
docker compose -p deployboard-stage30 \
  -f compose.prod28.yaml -f compose.stage30.yaml \
  up -d --no-deps api
python3 tools/release30_probe.py --url https://localhost:9443 \
  --ca-file ./stage30-ca.crt --private-dir private/stage30
```

Acceptance proof: (1) an existing staging record ID is readable after API recovery; (2) the failed version produced a nonzero exit or unhealthy status; (3) restored readiness and authenticated behavior pass; (4) a background job still reaches its expected terminal state; (5) staging PostgreSQL/Redis named volumes were not replaced. If any condition fails, record HOLD and investigate. Passing this drill does not establish that a real production rollback is safe after destructive migrations.

File 30.6 — tests/test_release30.py: complete offline safety tests

```
"""Offline release-safety tests (do not require a registry or Docker)."""
from pathlib import Path
from tempfile import TemporaryDirectory
import json
import unittest
import yaml
from tools.release30 import create_plan, decide, is_digest

P = "registry.example.org/deployboard/api@sha256:" + "a" * 64
Q = "registry.example.org/deployboard/api@sha256:" + "b" * 64

class ReleaseTests(unittest.TestCase):
    def test_valid_digest(self):
        self.assertTrue(is_digest(P))
        self.assertFalse(is_digest("registry.example.org/deployboard/api:latest"))
```

```
    def test_refuses_same_image(self):
        with TemporaryDirectory() as d:
            archive=Path(d)/"test.dump"; archive.write_bytes(b"test-only")
            with self.assertRaises(ValueError):
                create_plan(P, P, archive, "local")
    def test_backup_required(self):
        with self.assertRaises(ValueError):
            create_plan(P, Q, "/does/not/exist", "local")
    def test_holds_until_evidence_complete(self):
        with TemporaryDirectory() as d:
            archive=Path(d)/"test.dump"; archive.write_bytes(b"test-only")
            plan=create_plan(P, Q, archive, "local")
            self.assertEqual(plan["state"], "HOLD")
            with self.assertRaises(ValueError): decide(plan, "GO", "ready")
            plan["evidence"]={k:"PASS" for k in plan["evidence"]}
            self.assertEqual(decide(plan,"GO","manual review passed")["state"],"GO")
    def test_stage_is_isolated(self):
        root=Path(__file__).resolve().parents[1]
        stage=yaml.safe_load((root/"compose.stage30.yaml").read_text())
        fault=yaml.safe_load((root/"compose.fault30.yaml").read_text())
        self.assertTrue(all(v["file"].startswith("./private/stage30/")
                            for v in stage["secrets"].values()))
        self.assertTrue(all(s not in stage["services"] for s in ("db", "redis")))
        self.assertFalse("ports" in fault["services"]["api"])
        self.assertIn("42", " ".join(fault["services"]["api"]["command"]))

if __name__ == "__main__": unittest.main()
```

Command 30.11 — Validate the source-only release safeguards

```
python3 -m unittest tests.test_release30 -v
python3 tools/release30.py --help
# Expected: all release30 offline tests pass; no Docker changes.
# Keep the staging environment for review until evidence is captured.
```

RELEASE EXIT CRITERIA Release is approved only when the candidate is traceable, recovery evidence is current, staged behavior matches the contract, the production change has an accountable owner, and the post-deployment proof is recorded. A recovery is complete only after both user-facing service and durable records are verified.

Chapter 31 — Operate Docker Hosts and Respond to Incidents

31.1 Create a Practical Operator Runbook

A healthy deployment is not the same as an operable deployment. DeployBoard now has an HTTPS edge, authenticated API, PostgreSQL records, Redis Streams, a publisher, and a worker. A normal day requires evidence of capacity, backups, credentials, and release identity; a bad day requires a disciplined response that does not turn one failed service into lost records. This chapter gives you both routines using the exact Compose project established in Chapters 28–30.

OUTCOME FOR THIS CHAPTER You will create a repeatable operator checklist, distinguish host failure from application failure, verify capacity and TLS exposure, rehearse one dependency outage in an isolated project, and preserve a complete recovery record.

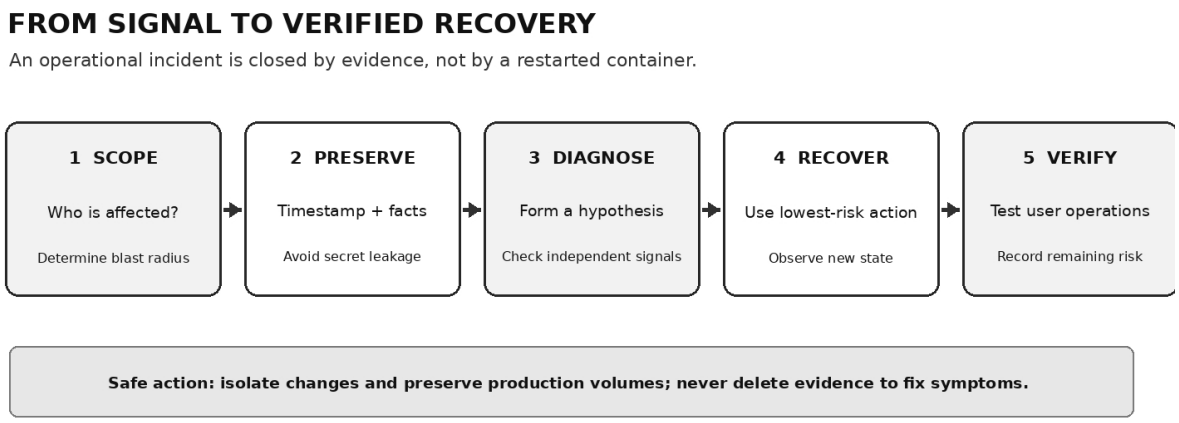


Figure 31.1 — An incident is closed by user-visible proof and preserved evidence—not by an unexamined restart.

Table 31.1 — First-response matrix

Signal	First safe check	Escalate when
HTTPS unavailable	Check DNS/TLS, proxy status, host reachability	Public users affected or cert invalid
API 503 / readiness fails	Check API readiness, DB health, published image	Writes failing or data consistency uncertain
Jobs remain pending	Check Redis/worker health and job ledger	Backlog grows beyond agreed delay
Disk or memory warning	Record free bytes, inodes, container resource use	Critical threshold or new errors
Unexpected privilege event	Restrict access, preserve minimal evidence	Possible host/secret compromise

The production identity is deployboard-prod28 and uses compose.prod28.yaml plus exactly one of compose.local28.yaml or compose.public28.yaml. In a Linux shell on the approved host, define a short, explicit command function. It changes no running resources by itself. Read-only observations are the default; do not combine them with a fault-injection override.

Command 31.1 — Define the correct production observation context

```
# Run from the DEPLOYBOARD project root on an APPROVED host.
# Choose local28 OR public28, never both.
export DEPLOYBOARD_IMAGE='REGISTRY/OWNER/deployboard@sha256:REAL_DIGEST'
dc() {
  docker compose -p deployboard-prod28      -f compose.prod28.yaml -f compose.local28.yaml "$@"
}
dc ps
dc config --images
# Production operations require Docker privileges; restrict their use.
```

CRITICAL PROJECT IDENTITY RULE The production stack is deployboard-prod28. An isolated simulator in Section 31.11 is deployboard-ops31. Never substitute the production name into the simulation commands; Compose project names determine container and volume isolation.

31.2 Monitor Disk, Logs, Containers, and Host Capacity

A Docker host can exhaust disk space through logs, pulled images, build cache, database growth, or backup archives long before an application reaches an obvious error. Measure capacity as trends, not a single snapshot. Record byte capacity and inode availability on the filesystem holding

Docker data, not only on the root mount. Use thresholds as provisional guardrails and tune them to your own growth and recovery window.

Command 31.2 — Read-only operational measurements

```
date -u +'%Y-%m-%dT%H:%M:%SZ'
findmnt /var/lib/docker          # If rootful Docker uses this path
df -h / /var/lib/docker
df -i / /var/lib/docker
free -h
uptime
docker system df
docker stats --no-stream
dc ps
# Example log window; review locally before sharing incident data.
dc logs --since 10m --tail 50 api publisher worker proxy
```

Table 31.2 — Example alert rules to calibrate

Signal	Warning starting point	Critical response
Filesystem used	80% sustained	90% or insufficient backup/recovery headroom
Inodes used	80% sustained	90% or new write failures
API readiness	2 consecutive failed checks	Failed user actions or widespread 5xx
Queue lag	Above normal baseline	Business processing deadline at risk
TLS validity	21 days remaining	Renewal failed / certificate expired

Docker recommends the local logging driver or explicit rotation to avoid unbounded log growth. DeployBoard already uses local logging with 10 MB files and a count of three for the production services. Changing the daemon-wide default does not retroactively update existing containers; record each container’s actual logging driver before you claim the policy is active.

Command 31.3 — Read-only log policy and size checks

```
docker info --format '{{.LoggingDriver}}'
# Review only a container ID returned for this project:
dc ps -q api
# Replace CONTAINER_ID with the returned ID.
docker inspect --format '{{json .HostConfig.LogConfig}}' CONTAINER_ID
# Avoid editing files directly under Docker's internal log directory.
# DO NOT run docker system prune --volumes to make room.
```

The following standard-library utility can run without Docker. Its exit code is 2 at the example critical 90% disk threshold; use the JSON in your

monitoring, but do not claim this one metric covers memory pressure, filesystem quotas, or inode exhaustion.

File 31.1 — ops31/host_capacity.py (complete read-only host check)

```
"""Read-only, standard-library-only Linux host capacity snapshot."""
import argparse, json, os, shutil
from datetime import datetime, timezone

def severity(used, warning=80, critical=90):
    return 'critical' if used >= critical else 'warning' if used >= warning else 'ok'

def disk_snapshot(path):
    t,u,f=shutil.disk_usage(path)
    pct=round(100*u/t,1) if t else 0.0
    return {'path':path,'size_gib':round(t/2**30,2),
            'free_gib':round(f/2**30,2),'used_pct':pct,'level':severity(pct)}

def main():
    ap=argparse.ArgumentParser()
    ap.add_argument('--path',default='/')
    args=ap.parse_args()
    if not os.path.isdir(args.path):
        ap.error('Path not found: '+args.path)
    report={'utc':datetime.now(timezone.utc).isoformat(),
            'cpu_count':os.cpu_count(),'disk':disk_snapshot(args.path)}
    print(json.dumps(report,indent=2))
    return 2 if report['disk']['level']=='critical' else 0
if __name__=='__main__': raise SystemExit(main())
```

Command 31.4 — Collect one host-capacity reading

```
python3 ops31/host_capacity.py --path /
# If Docker data is on another mount, point --path to that mount.
# Exit 0 means below critical threshold, NOT overall host approval.
```

31.3 Understand Daemon Downtime and Restart Behavior

An unhealthy container, a stopped Docker daemon, and a reboot are different events. restart: unless-stopped on DeployBoard services affects container restart policy; it does not repair a failed database query or guarantee that dependencies recover in the right order. A healthcheck can report unhealthy without automatically replacing its container. Compose service_healthy controls dependency startup, not continuous failover orchestration.

Docker live-restore is a distinct daemon feature. If deliberately enabled and supported, it can keep running Linux containers alive during some daemon interruptions; it is not a promise of uninterrupted connectivity or safe major upgrades. Logger pipes can eventually block if a stopped daemon cannot

drain output, and service management remains unavailable until the daemon returns.

Command 31.5 — Diagnose daemon state without initiating downtime

```
sudo systemctl is-active docker
sudo systemctl status docker --no-pager -n 20
sudo journalctl -u docker --since '15 min ago' --no-pager
# Display live-restore value (if Docker is reachable).
docker info --format '{{.LiveRestoreEnabled}}'
dc ps
# Never stop the Docker service on a production host as a test.
```

DECISION POINT For a daemon failure, stabilize the host, preserve journald evidence, restore daemon control under an approved maintenance procedure, then verify TLS, API readiness, database records, and workers. A daemon returning to active is not equivalent to an application recovery.

31.4 Patch Hosts, Engines, Images, and Dependencies Safely

Patch at four distinct layers: the Linux host/kernel, Docker Engine and Compose plugin, image bases and packages, and application dependencies. They do not share one update command or failure mode. A security patch to an image requires building, reviewing, and publishing a new digest (Chapters 20, 25, and 26); pulling the same digest cannot install different package bytes.

Table 31.3 — Patch plan by layer

Layer	Change method	Must be checked afterward
Linux kernel / host	Vendor package update; reboot only in window	SSH access, firewall, daemon, mounts
Docker Engine / Compose	Vendor-supported update path	Daemon control, Compose compatibility, service health
Runtime image	CI-tested rebuild and immutable digest	SBOM/scan, app behavior, TLS, jobs
PostgreSQL / Redis	Version- and data-format-specific upgrade plan	Restorable backup, compatibility, persistence

Before any change, obtain a current Chapter 29 PostgreSQL backup and recovery rehearsal evidence, a previous approved application image digest, a planned outage window, and a rollback owner. Review PostgreSQL and Redis version-specific upgrade notes. Upgrading a database image is not a generic replacement operation: on-disk version incompatibility can prevent

startup. Use Chapter 30's staging-first, expand-before-contract strategy for schema changes.

Command 31.6 — Patch-window preflight (no updates performed)

```
date -u +'%Y-%m-%dT%H:%M:%SZ'
sudo apt-get -s upgrade          # Ubuntu preview; inspect output
docker version --format '{{.Server.Version}}'
docker compose version
dc config --images
dc ps
docker system df
# Follow Chapters 29 and 30 for backup proof and approved rollout.
# Do not run unattended major Docker/DB upgrades on this host.
```

31.5 Handle Expiring Certificates and Credential Rotation

Caddy manages public ACME certificates and local-CA certificates, but automation depends on correct DNS, ports, outbound reachability, and a persistent writable /data directory. The Chapter 28 Compose stack deliberately retains caddydata and caddyconfig named volumes. Do not prune, reinitialize, or casually copy that state: it contains certificate keys and local trust material. A self-signed local CA is not globally trusted; clients must explicitly trust the right root.

Confirm not just a configured domain but the certificate the actual HTTPS endpoint serves. This Python check uses standard TLS hostname verification and supports a trusted local CA through --cafile. An expired or untrusted certificate returns a nonzero code rather than suggesting insecure verification bypasses.

File 31.2 — ops31/tls_probe.py (complete TLS-validity check)

```
"""Check live certificate with normal hostname verification; never disable TLS checks."""
import argparse, json, socket, ssl
from datetime import datetime, timezone

def remaining_days(not_after, now=None):
    now = now or datetime.now(timezone.utc)
    expires=datetime.fromtimestamp(ssl.cert_time_to_seconds(not_after),tz=timezone.utc)
    return (expires-now).total_seconds()/86400

def main():
    ap=argparse.ArgumentParser()
    ap.add_argument('hostname')
    ap.add_argument('--port',type=int,default=443)
    ap.add_argument('--cafile',help='Trust a local CA certificate when testing local HTTPS')
    ap.add_argument('--warn-days',type=int,default=21)
    args=ap.parse_args()
    if args.port < 1 or args.port > 65535: ap.error('Invalid TCP port')
    context=ssl.create_default_context(cafile=args.cafile)
    with socket.create_connection((args.hostname,args.port),timeout=8) as sock:
        with context.wrap_socket(sock,server_hostname=args.hostname) as tls:
```

```
        cert=tls.getpeercert()
    days=remaining_days(cert['notAfter'])
    report={'host':args.hostname,'port':args.port,'days_remaining':round(days,1),
        'expires_utc':cert['notAfter'],'warning':days<args.warn_days}
    print(json.dumps(report,indent=2))
    return 2 if days<0 else (1 if days<args.warn_days else 0)
if __name__=='__main__': raise SystemExit(main())
```

Command 31.7 — Verify a real public certificate or local CA

```
# Public DNS: replace with YOUR approved public hostname.
python3 ops31/tls_probe.py app.example.com --warn-days 21
# Chapter 28 local HTTPS, with its approved root CA in a safe path:
python3 ops31/tls_probe.py localhost --port 8443   --cafile ./approved-prod-ca.crt --warn-days 21
# No --insecure option: invalid TLS must fail the check.
```

Rotate viewer/operator credentials with client coordination and a tested secret-file update. In the established Chapter 19 design, a changed host file is not sufficient proof that an already-created container sees the new file. Recreate only the affected API service after the new token is secured and distributed through your approved channel. Confirm that the old token fails, new token works, and role permissions remain enforced. Database password rotation requires a separate, transaction-aware database procedure and coordination with all clients—not merely editing db_password.txt.

Command 31.8 — Approved API-token rotation sequence (outline)

```
# Maintenance window; prepare all dependent clients first.
python3 tools/rotate_ch19_token.py operator
# File is replaced securely; coordinate client cutover.
dc up -d --no-deps api
# Verify new operator access and old-token denial with the
# Chapter 28 smoke tests. Do NOT print tokens into incident logs.
```

31.6 Troubleshoot Registry Outages and Image Pull Failures

A deployment can fail to pull even while existing containers run normally. Separate DNS failure, network timeouts, registry-service degradation, incorrect repository paths, token permission, rate limits, architecture mismatch, and an absent digest. Treat an authentication failure as a credential problem; retry loops will not correct a missing scope. Do not promote a mutable latest tag as an outage workaround.

Command 31.9 — Read-only registry and image identity diagnosis

```
# The digest should already be recorded in your release manifest.
echo "$DEPLOYBOARD_IMAGE"   # Reference only; never credentials
docker image inspect "$DEPLOYBOARD_IMAGE"   --format '{{.Id}}'          # Fails if the exact image is not local
getent hosts registry-1.docker.io  # Example DNS diagnostic
# Check a pull only in an approved maintenance/staging window:
# docker pull "$DEPLOYBOARD_IMAGE"
# Do not log in with tokens on a command line or share auth files.
```

If the exact approved digest is already present locally, the running release may continue without a registry. That is not evidence that a new host can recover or that new instances can be started after local cache eviction. Record the digest, platform, and local cache state, declare a deployment HOLD, preserve known-good containers, and reattempt publication only after connectivity and authentication are understood.

31.7 Recognize Network Outages and Unhealthy Dependencies

User-visible 5xx errors require a route-level diagnosis: client DNS → TLS and Caddy → API → database or Redis → background worker. The API can remain live while Redis is unavailable; DeployBoard readiness chiefly reflects the database-backed serving path, so queue completion must be checked separately. A successful container-to-container DNS lookup does not prove that the upstream process is listening, and a successful TCP connection does not prove the correct records are returned.

Command 31.10 — Follow a failed request through the stack

```
# Local mode; use the trusted CA established in Chapter 28.
curl --cacert ./approved-prod-ca.crt -i https://localhost:8443/ready
dc ps
dc logs --since 10m --tail 50 proxy api worker redis db
# Inspect the actual Redis health and worker backlog via Ch. 22 tools.
# A DB issue: check PostgreSQL readiness and migration status first.
# Avoid printing database credentials or entire environment values.
```

AVOID CREATING A SECOND OUTAGE Do not restart all services simultaneously just because one dependency is unhealthy. Identify the failing boundary, verify last-known-good state, and correct the smallest component consistent with the evidence. Never delete PostgreSQL or Redis named volumes as a network troubleshooting step.

31.8 Review Security Incidents and Preserve Useful Evidence

Security incident response prioritizes containment and trustworthy evidence. A suspected Docker socket exposure or credential compromise may grant host-level control; use your organization’s incident plan, restrict unauthorized access, and involve the security owner. On a compromised host, local logs and executable tools may be untrustworthy. Capture remote or independently preserved evidence where supported, and keep access controls and timestamps intact.

Table 31.4 — Record evidence without spreading secrets

Preserve	Avoid	Why
UTC timeline and affected user actions	Guessing root cause immediately	Separates observation from hypothesis
Image digest, release decision, container state	Unredacted docker inspect/env dumps	Could expose credentials
Scoped logs with approved redaction	Copying all /var/lib/docker or TLS keys	Least data necessary for analysis
Backup/hash manifest and approved response	Deleting volumes or log files	Protects recovery and chain of custody

The simulation’s evidence collector intentionally captures only its own benign services and a bounded docker system df summary. It creates owner-only local JSON evidence and truncates output. Do not repurpose it to collect real production logs unless you implement a reviewed redaction and secure archival procedure appropriate to your environment.

File 31.3 — ops31/capture_sim.py (complete simulation-only evidence capture)

```
"""Capture only the isolated simulation's evidence; no production logs or secrets."""
import argparse, json, os, subprocess
from pathlib import Path
from datetime import datetime, timezone

PROJECT='deployboard-ops31'
COMPOSE='ops31/compose.sim31.yaml'
def safe_name(s):
    return ''.join(ch for ch in s if ch.isalnum() or ch in '-_')[:48]
def capture(argv, limit=12000):
    try:
        p=subprocess.run(argv,text=True,stdout=subprocess.PIPE,
              stderr=subprocess.STDOUT,timeout=20,check=False)
        return {'command':' '.join(argv),'returncode':p.returncode,
                'output':p.stdout[:limit]}
    except (FileNotFoundError,subprocess.TimeoutExpired) as e:
        return {'command':' '.join(argv),'error':type(e).__name__}
def main():
    ap=argparse.ArgumentParser()
    ap.add_argument('--label',default='incident')
    ap.add_argument('--directory',default='ops31/evidence')
    args=ap.parse_args()
    p=Path(args.directory).resolve()
    p.mkdir(mode=0o700,parents=True,exist_ok=True)
    os.chmod(p,0o700)
    stamp=datetime.now(timezone.utc).strftime('%Y%m%dT%H%M%SZ')
    out=p/(stamp+'_'+safe_name(args.label)+'.json')
    base=['docker','compose','-p',PROJECT,'-f',COMPOSE]
    result={'utc':stamp,'scope':'isolated simulator ONLY','observations':[
        capture(base+['ps','--all']),
        capture(base+['logs','--since','10m','--tail','40','gateway','upstream']),
        capture(['docker','system','df'])]}
    with os.fdopen(os.open(out,os.O_WRONLY|os.O_CREAT|os.O_EXCL,0o600),'w') as f:
        json.dump(result,f,indent=2)
    print('Evidence saved (private local file):',out)
```

```
if __name__=='__main__': main()
```

EVIDENCE FORM Incident ID: _____ UTC detected: _____ First affected action: _____ Affected services: _____ Approved owner: _____ Hypothesis and contrary evidence: _____ Recovery action: _____ Post-recovery proof: _____ Remaining risk / follow-up: _____

31.9 Plan Capacity Upgrades and Scaling Limits

Scaling the number of containers is not the same as increasing useful throughput. First identify the resource that saturates: CPU, memory, disks, PostgreSQL connections, Redis queue lag, or outbound bandwidth. Compare peak load and recovery headroom with the Chapter 24 benchmarks. Every worker replica increases potential concurrency and database pool pressure; the Chapter 15 idempotency design must remain correct under redelivery and duplicate execution before scaling worker count.

A simple host-sizing decision uses workload peaks, headroom for failure recovery, and backup growth. For example, 70% peak RAM use on a host with little page cache does not justify a promise of twofold throughput when PostgreSQL storage IOPS is already saturated. Prefer measured bottleneck removal to speculative CPU upgrades. Leave enough disk for live database growth, temporary backup archives, retained images, and logs.

Command 31.11 — Gather capacity evidence before resizing

```
python3 ops31/host_capacity.py --path /
docker stats --no-stream
docker system df
# Collect P95 latency, failed requests, job lag and DB metrics
# using Chapter 22 / 24 instruments at the same workload.
# Record peak observation window and host resource headroom.
```

31.10 Know When One-Host Compose Is No Longer Enough

Compose is an excellent single-host application manager, but it is not a multi-host high-availability scheduler. Your single Linux host remains a failure domain for the proxy, API, database, and local named volumes. Restart policies do not protect against server loss; a verified backup reduces data-loss risk but does not provide instantaneous failover. A separate recovery site, external database, replicated storage, and orchestrated multi-

host scheduling require their own architecture, operating costs, and disaster drills.

Table 31.5 — Scaling decision, not a product popularity contest

Observed requirement	Single-host Compose is enough when	Reconsider architecture when
Availability	Planned maintenance is acceptable	Required uptime exceeds one-host resilience
Recovery time	Manual restore meets tested RTO	Failover must be automatic and rapid
Traffic/load	One host has proven capacity headroom	Saturation remains after tuning
Team operations	Small trusted admin group	Many teams need policy-driven deployment

Before changing orchestration platforms, write down target recovery and availability objectives, data semantics, certificate ownership, observability requirements, migration cost, and who will operate the next system. A move to Kubernetes or managed services can solve some failure-domain problems while creating new security and operational risks; do not treat the transition itself as a reliability guarantee.

31.11 Checkpoint: Complete an Operations Simulation

The final rehearsal isolates the decision making from any live database. You will start a two-service simulator on 127.0.0.1:18031, prove that its gateway is live and its dependency is ready, stop only the simulator’s upstream, capture an HTTP 503 for readiness while liveness remains 200, preserve evidence, restore the upstream, and verify recovery. The production project deployboard-prod28 is never used in the fault commands.

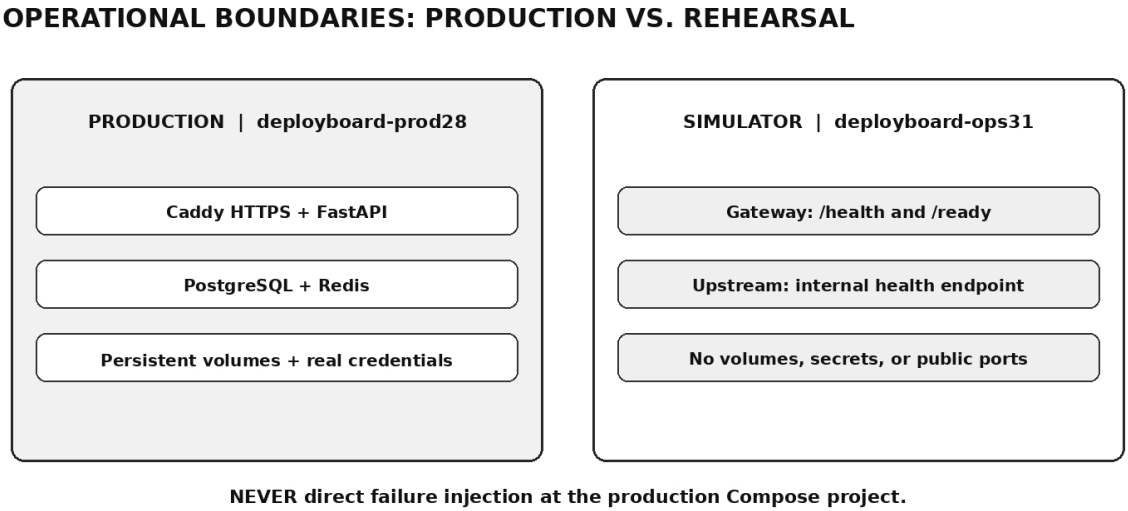


Figure 31.2 — The isolated simulator has no production credentials, named volumes, privileged mounts, or public binding.

File 31.4 — ops31/compose.sim31.yaml (complete fault-simulation Compose)

```
# Isolated rehearsal. It never mounts or names DeployBoard production volumes.
name: deployboard-ops31
services:
  upstream:
    image: python:3.12-alpine
    command: ["python", "-u", "/srv/upstream.py"]
    volumes: ["./upstream.py:/srv/upstream.py:ro"]
    networks: [isolated]
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8090/health',timeout=2)"]
      interval: 5s
      timeout: 3s
      retries: 3
    restart: unless-stopped
    logging: &small_log
      driver: local
      options: {max-size: "5m", max-file: "2"}
  gateway:
    image: python:3.12-alpine
    command: ["python", "-u", "/srv/gateway.py"]
    volumes: ["./gateway.py:/srv/gateway.py:ro"]
    networks: [isolated]
    ports: ["127.0.0.1:18031:8080"]
    depends_on:
      upstream: {condition: service_healthy}
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8080/health',timeout=2)"]
      interval: 5s
      timeout: 3s
      retries: 3
    restart: unless-stopped
    logging: *small_log
networks:
  isolated: {driver: bridge}
```

File 31.5 — ops31/upstream.py (complete dependency service)

```
"""Dependency simulator: no secrets, database, or external port."""
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
```

```
import json

class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        status = 200 if self.path == '/health' else 404
        blob = json.dumps({'service':'upstream','status':'ok' if status==200 else 'missing'}).encode()
        self.send_response(status)
        self.send_header('Content-Type','application/json')
        self.send_header('Content-Length',str(len(blob)))
        self.end_headers()
        self.wfile.write(blob)
    def log_message(self, fmt, *args):
        return  # avoid request information in lab logs

if __name__ == '__main__':
    ThreadingHTTPServer(('0.0.0.0',8090),Handler).serve_forever()
```

File 31.6 — ops31/gateway.py (complete gateway with distinct liveness/readiness)

```
"""Gateway: liveness 200 can coexist with readiness 503."""
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
from urllib.request import urlopen
from urllib.error import URLError, HTTPError
import json, sys

def upstream_ready():
    try:
        with urlopen('http://upstream:8090/health',timeout=1.5) as r:
            return r.status == 200
    except (URLError, HTTPError, TimeoutError, OSError):
        return False

def assessment(path, ready):
    if path == '/health':
        return (200, {'service':'gateway','live':True})
    if path == '/ready':
        return (200 if ready else 503,
                {'service':'gateway','ready':ready,'dependency':'upstream'})
    return (404, {'error':'not found'})

class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        status, data = assessment(self.path, upstream_ready() if self.path=='/ready' else True)
        blob=json.dumps(data,separators=(',',':')).encode()
        self.send_response(status)
        self.send_header('Content-Type','application/json')
        self.send_header('Content-Length',str(len(blob)))
        self.end_headers()
        self.wfile.write(blob)
        print(json.dumps({'service':'gateway','route':self.path,'status':status}),flush=True)
    def log_message(self, fmt, *args):
        return

if __name__ == '__main__':
    ThreadingHTTPServer(('0.0.0.0',8080),Handler).serve_forever()
```

Command 31.12 — Observe, break, verify, recover; SIMULATION ONLY

```
# Run from DeployBoard project root with Docker Engine available.
sim() {
  docker compose -p deployboard-ops31     -f ops31/compose.sim31.yaml "$@"
}
sim config -q
sim up -d
curl --fail-with-body http://127.0.0.1:18031/ready
python3 ops31/capture_sim.py --label baseline
```

```
# Controlled dependency outage: only the isolated project.
sim stop upstream
curl -i http://127.0.0.1:18031/ready    # EXPECT HTTP 503
curl --fail-with-body http://127.0.0.1:18031/health
python3 ops31/capture_sim.py --label dependency-down

sim start upstream
# If the upstream has not finished starting, repeat the probe.
curl --fail-with-body http://127.0.0.1:18031/ready
python3 ops31/capture_sim.py --label recovered
sim down      # No named volumes exist in this simulation.
# Never add -v, and never substitute deployboard-prod28.
```

Expected evidence: baseline readiness 200; after the simulated upstream stop, readiness 503 and liveness 200; after recovery, readiness 200 again. The gateway's own container remains healthy throughout because its healthcheck uses /health, deliberately demonstrating why liveness alone can hide dependency failures. If Docker Engine is not available, these outcomes are acceptance criteria, not results you may claim to have observed.

File 31.7 — tests/test_ops31.py (complete offline regression tests)

```
import unittest
from datetime import datetime, timezone
from pathlib import Path
import yaml
from ops31.gateway import assessment
from ops31.host_capacity import severity
from ops31.tls_probe import remaining_days
from ops31.capture_sim import safe_name

ROOT=Path(__file__).resolve().parents[1]
class Ops31Tests(unittest.TestCase):
    def test_gateway_live_when_downstream_fails(self):
        self.assertEqual(assessment('/health', False)[0],200)
        self.assertEqual(assessment('/ready', False)[0],503)
    def test_gateway_ready_when_upstream_is_ok(self):
        self.assertEqual(assessment('/ready', True)[0],200)
    def test_thresholds(self):
        self.assertEqual([severity(v) for v in (79,80,89.9,90)],
                         ['ok','warning','warning','critical'])
    def test_expired_certificate_is_negative(self):
        self.assertLess(remaining_days('Jan  1 00:00:00 2020 GMT',
                                       datetime(2026,10,20,tzinfo=timezone.utc)),0)
    def test_evidence_filename_traversal_is_removed(self):
        self.assertNotIn('/',safe_name('../production/private'))
    def test_simulation_is_isolated_and_local_only(self):
        data=yaml.safe_load((ROOT/'ops31/compose.sim31.yaml').read_text())
        self.assertIn('127.0.0.1:18031:8080',data['services']['gateway']['ports'])
        self.assertNotIn('ports',data['services']['upstream'])
        self.assertNotIn('volumes',data)
        self.assertEqual(set(data['services']),{'gateway','upstream'})
if __name__=='__main__':unittest.main()
```

Command 31.13 — Test the supplied utilities without Docker

```
python3 -m unittest tests.test_ops31 -v
python3 ops31/host_capacity.py --path /
python3 ops31/tls_probe.py --help
# Offline tests exercise code and safety constraints, not Docker behavior.
```

FINAL OPERATIONS GATE PASS only with a dated runbook, documented service owners, measured capacity and log limits, current verified backup-and-restore evidence, trusted TLS status, release digest and rollback record, complete simulation evidence, and a signed response owner. If a condition is untested, mark it PENDING—not PASS.

Chapter 32 — Capstone: Ship and Operate a Secure DeployBoard Release

32.1 Rebuild the Application from a Clean Workspace

DeployBoard is no longer a sequence of isolated Docker demonstrations. It is one application with an authenticated FastAPI service, a durable PostgreSQL deployment ledger, Redis-backed asynchronous checks, a publisher, a worker, and a Caddy HTTPS entry point. The capstone has a different purpose from an installation tutorial: you will prove that another operator can reconstruct, ship, observe, and recover one identified release using only the files and methods supplied throughout this book.

RELEASE CONTRACT A passing capstone needs a clean source tree, independently isolated tests, recorded image digest, HTTPS proof, preserved data, a tested recovery route, and an approval signed against observed evidence. A successful container start is insufficient.

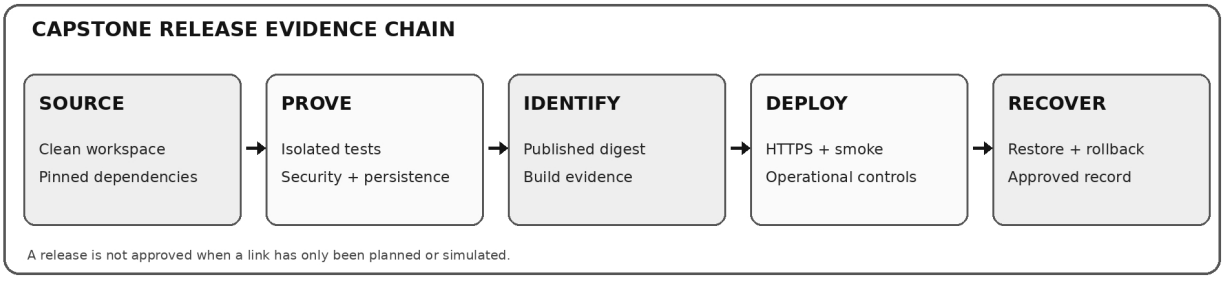


Figure 32.1 — Each link in the evidence chain must be verified, not merely configured.

Use the cumulative Chapter 32 source tree supplied with this book. It contains the source from Chapters 13–31, with the new ops32 tools and tests. You need a Linux Docker Engine with the Compose and Buildx plugins for the full lab, Python 3 for offline utilities, and either a local Linux VM or an approved remote host. No GitHub repository is required for the local path. The optional publishing and CI branches use your own OCI registry and account.

Command 32.1 — Start clean and inspect the complete source tree

```
# Copy or extract the cumulative DeployBoard source into a NEW workspace.
# On Linux, from the extracted project directory:
pwd
```

```
python3 -m unittest discover -s tests -p 'test_capstone32.py' -v
find app migrations ops31 ops32 proxy tests tools -type f | sort
ls -la Dockerfile requirements.txt compose.prod28.yaml
# Do not copy private/, qa-private/, backups29/ or *.dump into a registry image.
# Recreate production credentials ONLY by an approved, host-managed process.
```

Do not rename the production Compose project accidentally. The Chapter 28 production identity is deployboard-prod28, while the Chapter 21 QA quality gate is deployboard-qa21. They must never share database volumes or fault-injection controls. Work from a fresh directory but preserve the intended project identity when you deliberately update an existing host; that identity affects volume names. On a new host, do not assume an empty volume contains your previous records.

32.2 Verify the Dockerfiles and Pinned Dependencies

An image tag describes a convenient moving reference, not an immutable base. The inherited Dockerfile still begins with python:3.13-slim; the Python package versions in requirements.txt are exact pins, but the base image is not yet frozen. Fix that gap for the capstone release rather than calling a floating Dockerfile “reproducible.” The helper below creates a separate release Dockerfile and refuses to overwrite an existing one.

Command 32.2 — Inspect a real upstream base manifest and choose a digest

```
docker buildx imagetools inspect python:3.13-slim
# Record the actual sha256 manifest/index digest for your target platform.
# Replace the placeholder with the REAL reviewed digest, not a fabricated one.
python3 ops32/pin_base.py --base   'python:3.13-slim@sha256:REPLACE_WITH_64_LOWERCASE_HEX'
# The script deliberately rejects the placeholder until it is replaced.
docker build --check -f Dockerfile.release32 .
# If your installed Docker does not support --check, use Buildx
# build validation and review the generated Dockerfile explicitly.
```

File 32.1 — ops32/pin_base.py (complete release-base pinning utility)

```
"""Replace only the base-image line in a release-specific Dockerfile.

Usage: python3 ops32/pin_base.py --base python:3.13-slim@sha256:<64hex>
The resolved digest must be inspected and approved first. No network access here.
"""
import argparse
from pathlib import Path
import re

ROOT = Path(__file__).resolve().parents[1]
BASE = re.compile(r"^python:3\.13-slim@sha256:[0-9a-f]{64}$")


def render(source: str, base: str) -> str:
    if not BASE.fullmatch(base):
        raise ValueError("Expected an approved python:3.13-slim digest reference")
    lines = source.splitlines(keepends=True)
```

```
    if not lines or lines[0].strip() != "FROM python:3.13-slim":
        raise ValueError("Source Dockerfile base line has changed; review it first")
    lines[0] = f"FROM {base}\n"
    return "".join(lines)


def main(argv=None):
    parser = argparse.ArgumentParser()
    parser.add_argument("--base", required=True)
    args = parser.parse_args(argv)
    source = (ROOT / "Dockerfile").read_text(encoding="utf-8")
    target = ROOT / "Dockerfile.release32"
    if target.exists():
        raise SystemExit("HOLD: Dockerfile.release32 exists; do not overwrite it")
    try:
        result = render(source, args.base)
    except ValueError as exc:
        raise SystemExit("HOLD: " + str(exc)) from None
    target.write_text(result, encoding="utf-8")
    print("Created Dockerfile.release32 with a resolved base-image digest")


if __name__ == "__main__":
    main()
```

Record the source and dependency decisions: exact package versions, the base reference and platform, whether the registry supports OCI attestations, and whether the application test results were collected from this same source. Pinning a digest does not eliminate vulnerabilities; retain the Chapter 20 scan and triage gate. Rebuilding later from the same files may not be bit-for-bit identical if your build environment or platform changes; digest verification, retained build metadata, and source control provide the stronger evidence trail.

32.3 Launch the Full Compose Stack and Database Schema

First prove a clean development or QA build in an isolated project, then deploy the published production image. These are deliberately separate actions. The Chapter 21 quality gate creates QA-only secret fixtures and uses the Compose project deployboard-qa21. It boots its own PostgreSQL and Redis, applies Alembic migrations before API tests, and checks the worker and application paths. Never aim its fault steps at production volumes.

Command 32.3 — Execute the isolated container quality gate

```
docker version
docker compose version
docker buildx version
python3 tools/quality_gate.py --cleanup
# This builds the QA service images, starts fresh QA services, migrates,
```

```
# executes the app and live stack tests, checks persistence and exit codes.
# --cleanup runs compose down WITHOUT -v; QA data can still remain.
# If the gate fails, preserve resources for investigation, then rerun.
```

If running on a truly fresh host, confirm no incompatible QA named volumes already exist. “Fresh source folder” alone does not imply fresh Compose data: volume identity depends on the project name. To reinitialize a disposable QA environment, explicitly review and archive any existing QA data first; never treat `down -v` as an ordinary cleanup shortcut. The production launch follows later and always runs the migration task before starting the API.

32.4 Prove That Data Survives Container Replacement

Use the quality gate’s persistence marker to distinguish container replacement from data deletion. Its `seed` phase writes a named record through the API; it then force-recreates only the API container and runs `verify`. PostgreSQL remains running with its named volume. This is the behavior you actually want to demonstrate: database records are not inside the API container filesystem.

Command 32.4 — Targeted QA persistence proof

```
# The quality_gate.py command performs these QA actions already.
# To inspect the isolated project after it runs:
docker compose -p deployboard-qa21   -f compose.yaml -f compose.harden.yaml -f compose.qa.yaml ps
# The verification marker is QA-CH21-PERSISTENCE-RECREATE.
# Review the quality gate output for SEED -> RECREATE API -> VERIFY.
# Never replace deployboard-qa21 with deployboard-prod28 here.
```

Evidence must include the original marker identity, the API recreation action, and the post-recreate record. Container IDs may change; the marker must not. Do not confuse this test with host failure, disk loss, or an actual backup restore—those are separate recovery claims assessed in Section 32.10.

32.5 Run Automated Tests and Security Checks

A passing test suite is only one release gate. Run the application tests, Compose configuration checks, privilege inspection, vulnerability scan, and registry evidence review as different controls. A Dockerfile lint check does not prove a container is non-root at runtime; an image scan does not prove

correct authentication; a passing API smoke test does not prove a database can be restored.

Command 32.5 — Quality, configuration and security evidence

```
python3 -m unittest discover -s tests -p 'test_capstone32.py' -v
python3 tools/ci26_validate.py      # install requirements-ci.txt first
python3 tools/host27_preflight.py --json  # run on the approved Linux host
# Run Chapter 21's full live quality gate before authorizing a release.
python3 tools/quality_gate.py --cleanup
# With a real candidate image stored locally:
trivy image --severity HIGH,CRITICAL --exit-code 1   'REGISTRY/OWNER/deployboard:VERSION'
# Review findings against Chapter 20's explicit exception policy.
# No sample vulnerability count is a measured result.
```

Table 32.1 — Do not collapse independent checks

Gate	What it proves	What it cannot prove
QA tests	Specified API and DB/queue behavior in test environment	Public TLS or remote-host readiness
Image scan	Known vulnerabilities in scanned image at scan time	Absence of all vulnerabilities or runtime exploitation
Hardening inspection	Configured user, capabilities and mount restrictions	All threat paths are eliminated
Restore drill	Recoverability of tested backup in test environment	Zero data loss in any future incident

The Chapter 26 workflow is a starting point, not a final security certification. Before using a public hosted pipeline for this capstone, review third-party action references and pin approved commits, protect trusted tags and environments, and ensure untrusted pull requests cannot obtain publishing or SSH credentials. The final candidate image should be built from the reviewed digest-pinned Dockerfile.release32; update your pipeline build target deliberately rather than silently continuing to publish from the earlier floating-base Dockerfile.

32.6 Confirm Secret Handling and Least-Privilege Controls

Production password and tokens are mounted as files under /run/secrets in application containers. Those files originate on the approved host and are excluded from the source package and build context. A running container is not a secret vault: root on the host, a privileged runtime, or a compromised process may still read mounted credentials. The objective is to reduce

accidental exposure and sharply limit access, not promise that secrets cannot be stolen.

Command 32.6 — Review controls without displaying credentials

```
# From the approved deployment host. NO cat of secret files.
stat -c '%a %n' private/*.txt    # review host permissions and ACLs
# Check that .gitignore and .dockerignore exclude private/:
git check-ignore private/db_password.txt  # only if using git locally
# Use one approved mode. This example is local HTTPS.
export DEPLOYBOARD_IMAGE='REGISTRY/OWNER/deployboard@sha256:REAL_DIGEST'
docker compose -p deployboard-prod28   -f compose.prod28.yaml -f compose.local28.yaml config -q
# During runtime, examine .HostConfig.ReadonlyRootfs, .CapDrop,
# .SecurityOpt, .NetworkMode and published ports with docker inspect.
# Never print `docker inspect .Config.Env` into a shared incident record.
```

Verify that API, publisher, and worker have a non-root process identity (from the image), read-only root filesystems, dropped capabilities, and `no-new-privileges`. Caddy retains only the bound-port capability. PostgreSQL and Redis use writable persistent volumes and should not have published host ports. The hardened overlays were validated separately in earlier chapters; repeat the inspection on the containers that actually run the approved release.

32.7 Publish an Identifiable, Versioned Release Image

Only after the quality and security gate passes should you publish a versioned image. Use a release name such as v1.0.0 and a deliberately selected target platform, not latest. Capture build metadata and inspect the pushed digest from the registry; do not invent a 64-character digest to make the commands look complete. If publication is not available, record the release as HOLD and complete the local-only functional path without claiming a distributed release.

Command 32.7 — Build, publish and resolve the real registry digest

```
# Set these in your shell to your own approved values:
export IMAGE_REPO='ghcr.io/YOUR_OWNER/deployboard'
export RELEASE_TAG='v1.0.0'
# Authenticate using a credential helper or password-stdin, not a CLI token.
mkdir -p evidence/capstone32
# Match --platform to your target Linux host architecture.
docker buildx build --platform linux/amd64   -f Dockerfile.release32   --provenance=true --sbom=true   --metadata-file
evidence/capstone32/build-meta.json   --push -t "${IMAGE_REPO}:${RELEASE_TAG}" .
docker buildx imagetools inspect "${IMAGE_REPO}:${RELEASE_TAG}"
# COPY the actual registry index/manifest digest from the output:
export DEPLOYBOARD_IMAGE='ghcr.io/YOUR_OWNER/deployboard@sha256:REAL_DIGEST'
# Do not proceed with placeholder text; pin to the observed digest.
```

A published tag can move, but an OCI reference at `@sha256:` names a specific manifest or index. For multi-platform indexes, record the target platform as well, because different hosts resolve different platform-specific images. Inspect the tag and pinned digest, and retain the SBOM and scan review with your release approval. The source receipt below hashes non-secret book-source files; it is not an image signature, a trusted attestation, or proof that the registry behaved honestly.

File 32.2 — ops32/release_receipt.py (complete source-linked evidence utility)

```
"""Record and check a non-secret, source-linked capstone release receipt.

This is a SHA-256 integrity record, NOT a cryptographic signature or
proof that a registry or deployment is trustworthy.
"""
import argparse
from datetime import datetime, timezone
import hashlib
import json
from pathlib import Path
import re
import sys

ROOT = Path(__file__).resolve().parents[1]
IMAGE = re.compile(r"^[a-z0-9][a-z0-9._:/-]*@sha256:[a-f0-9]{64}$")
SOURCES = (
    "Dockerfile.release32", "requirements.txt", "compose.prod28.yaml",
    "compose.local28.yaml", "compose.public28.yaml", "proxy/Caddyfile.local28",
    "proxy/Caddyfile.public28", "migrations/versions/0001_initial.py",
    "migrations/versions/0002_jobs.py", "ops32/pin_base.py",
    "ops32/release_receipt.py", "ops32/smoke32.py",
)


def is_digest_image(value: str) -> bool:
    return bool(IMAGE.fullmatch(value))


def sha(path: Path) -> str:
    h = hashlib.sha256()
    with path.open("rb") as file:
        for part in iter(lambda: file.read(1024 * 1024), b""):
            h.update(part)
    return h.hexdigest()


# Capture the complete *book-supplied source tree*, not generated secrets.
SOURCE_DIRS = {"app", "migrations", "tools", "tests", "proxy", "ops31",
               "ops32", "site", "review", "release", "faults", ".github"}
SOURCE_NAMES = {".gitignore", ".dockerignore", ".env.example",
                "alembic.ini", "pytest.ini", "recovery29_evidence_template.md",
                "release30_change_window.md"}


def source_digests(root: Path) -> dict[str, str]:
    result = {}
    for rel in SOURCES:
        path = root / rel
        if not path.is_file() or path.is_symlink():
            raise ValueError("Missing or symlinked required source: " + rel)
    for path in root.rglob("*"):
```

```
        if not path.is_file():
            continue
        rel = path.relative_to(root)
        parts = rel.parts
        if "__pycache__" in parts or path.suffix == ".pyc":
            continue
        root_name = parts[0]
        if not (root_name in SOURCE_DIRS or
                (len(parts) == 1 and (root_name in SOURCE_NAMES or
                 root_name.startswith(("Dockerfile", "compose.",
                                       "requirements", "CHAPTER_"))))):
            continue
        if path.is_symlink():
            raise ValueError("Symlink in source tree: " + rel.as_posix())
        result[rel.as_posix()] = sha(path)
    return dict(sorted(result.items()))


def receipt(root: Path, image: str, mode: str) -> dict:
    if not is_digest_image(image):
        raise ValueError("A full registry digest reference is required")
    if mode not in ("local", "public"):
        raise ValueError("TLS mode must be local or public")
    return {
        "schema": "deployboard-capstone32-v1",
        "recorded_utc": datetime.now(timezone.utc).isoformat(timespec="seconds"),
        "image": image,
        "https_mode": mode,
        "project": "deployboard-prod28",
        "source_sha256": source_digests(root),
        "approval": "PENDING: attach tested release, smoke, backup and rollback evidence",
    }


def matches(root: Path, data: dict) -> bool:
    try:
        if data["schema"] != "deployboard-capstone32-v1":
            return False
        if data["project"] != "deployboard-prod28":
            return False
        if not is_digest_image(data["image"]):
            return False
        if data["https_mode"] not in ("local", "public"):
            return False
        return data["source_sha256"] == source_digests(root)
    except (ValueError, KeyError, TypeError):
        return False


def main(argv=None) -> int:
    p = argparse.ArgumentParser()
    sub = p.add_subparsers(dest="action", required=True)
    make = sub.add_parser("record")
    make.add_argument("--image", required=True)
    make.add_argument("--mode", choices=["local", "public"], required=True)
    make.add_argument("--out", default="evidence/capstone32/receipt.json")
    check = sub.add_parser("verify")
    check.add_argument("--file", default="evidence/capstone32/receipt.json")
    a = p.parse_args(argv)
    try:
        if a.action == "record":
            data = receipt(ROOT, a.image, a.mode)
            target = (ROOT / a.out).resolve()
            allowed = (ROOT / "evidence" / "capstone32").resolve()
            if target.parent != allowed or target.suffix != ".json":
                raise ValueError("Receipt must be written under evidence/capstone32/")
```

```
        target.parent.mkdir(parents=True, exist_ok=True)
        if target.exists():
            raise ValueError("Receipt exists; archive it before a new release")
        target.write_text(json.dumps(data, indent=2) + "\n", encoding="utf-8")
        print("RECORDED source hashes and image reference; approval remains PENDING")
    else:
        data = json.loads((ROOT / a.file).read_text(encoding="utf-8"))
        if not matches(ROOT, data):
            raise ValueError("Receipt does not match the current sources")
        print("PASS: receipt source hashes match; NOT deployment approval")
    return 0
  except (OSError, ValueError, json.JSONDecodeError) as exc:
    print("HOLD: " + str(exc), file=sys.stderr)
    return 2


if __name__ == "__main__":
  raise SystemExit(main())
```

Command 32.8 — Write and verify a release receipt

```
python3 ops32/release_receipt.py record   --image "$DEPLOYBOARD_IMAGE" --mode local
python3 ops32/release_receipt.py verify
# It starts at approval PENDING; attach real scan/QA/digest/host evidence.
# A changed source file must cause the verification to fail.
```

32.8 Deploy over HTTPS to a Local VM or Optional Remote Server

Choose one network mode, not a mixture of both. The no-cost learning route uses an approved Linux VM and Caddy’s internal certificate authority with the proxy bound to the host’s loopback address, 127.0.0.1:8443. For an optional public server, first approve DNS, host firewall rules, and ownership of ports 80 and 443. Caddy then obtains publicly trusted certificates where eligibility and reachability requirements are met. Do not bypass certificate checks with `curl -k` to manufacture a PASS.

Command 32.9 — Migrate before starting the API; local HTTPS release

```
export DEPLOYBOARD_IMAGE='ghcr.io/YOUR_OWNER/deployboard@sha256:REAL_DIGEST'
# Create valid private/*.txt on the host using Chapter 19/27 guidance.
# Validate the host first; then request an explicit live deployment:
python3 tools/deploy28.py local   --image "$DEPLOYBOARD_IMAGE" --run
# It checks Compose, pulls the image, starts DB/Redis, migrates,
# then starts API, publisher, worker and Caddy; approval is separate.
# For public mode use 'public --domain app.YOUR_DOMAIN' and approved DNS.
# Do not run both local and public overrides in one Compose invocation.
```

Keep the local CA trust path explicit: export Caddy’s CA certificate by an approved procedure and import only that public CA certificate into your client trust store, or pass it as a trust file to the smoke utility. Never distribute the local CA private key. A remote VM with loopback-only proxy access can be reached through SSH port forwarding, but the SSH command

itself does not establish certificate trust; validate the hostname and trusted CA used by your HTTPS client.

32.9 Verify Health, Logs, Resource Controls, and User Flows

Run a read-only smoke test against the published HTTPS entry point. The test checks /health and /ready independently, refuses an unauthenticated list request, allows a viewer list, and rejects a viewer write. It reads token files without printing values. The optional staging-only write path creates a deployment, requests an asynchronous check, waits for successful worker completion, and deletes the fixture. Do not enable it on production because the cleanup may fail after a partial outage and because it mutates business records.

File 32.3 — ops32/smoke32.py (complete TLS/auth and staging-job smoke utility)

```
"""TLS-verified release smoke gate: no mutation unless --staging-write.

Run from a trusted machine with viewer/operator token FILES; no token CLI args.
"""
import argparse
import json
from pathlib import Path
import ssl
import sys
import time
from urllib.error import HTTPError, URLError
from urllib.parse import quote, urlsplit
from urllib.request import Request, urlopen
from uuid import uuid4


def safe_base(url: str) -> str:
    u = urlsplit(url)
    if u.scheme != "https" or not u.hostname or u.username or u.password or u.query or u.fragment:
        raise ValueError("Use an HTTPS base URL without credentials or query")
    return url.rstrip("/")


def call(base, path, ctx, token=None, method="GET", payload=None):
    headers = {"Accept": "application/json"}
    if token:
        headers["Authorization"] = "Bearer " + token
    data = None
    if payload is not None:
        data = json.dumps(payload).encode("utf-8")
        headers["Content-Type"] = "application/json"
    req = Request(base + path, data=data, headers=headers, method=method)
    try:
        with urlopen(req, timeout=8, context=ctx) as resp:
            raw = resp.read(65536)
            return resp.status, json.loads(raw) if raw else {}
    except HTTPError as exc:
        return exc.code, {}
```

```
def token_file(directory, name):
   path = Path(directory) / (name + ".txt")
   if path.is_symlink():
      raise ValueError("Refusing symlinked token file")
   value = path.read_text(encoding="utf-8").strip()
   if not value:
      raise ValueError("Token file is empty")
   return value


def run(base, ca_file, directory, staging_write=False):
   base = safe_base(base)
   ctx = ssl.create_default_context(cafile=ca_file)
   viewer = token_file(directory, "viewer_token")
   operator = token_file(directory, "operator_token")
   sample = {"name": "capstone32-" + uuid4().hex[:9],
             "environment": "staging", "version": "32.0"}
   checks = [
      ("liveness", "/health", None, "GET", None, 200),
      ("readiness", "/ready", None, "GET", None, 200),
      ("anonymous denied", "/v1/deployments", None, "GET", None, 401),
      ("viewer reads", "/v1/deployments", viewer, "GET", None, 200),
      ("viewer write denied", "/v1/deployments", viewer, "POST", sample, 403),
   ]
   for label, path, tok, verb, payload, expected in checks:
      status, _ = call(base, path, ctx, tok, verb, payload)
      if status != expected:
         print(f"HOLD: {label}, HTTP {status} (expected {expected})")
         return 2
      print("PASS: " + label)
   if not staging_write:
      print("PASS: read-only smoke gate; worker execution NOT yet proven")
      return 0
   # Only in an isolated staging environment. Cleanup after the job resolves.
   status, obj = call(base, "/v1/deployments", ctx, operator, "POST", sample)
   if status != 201 or not isinstance(obj.get("id"), int):
      print("HOLD: staging record creation did not succeed")
      return 2
   item = obj["id"]
   try:
      key = "capstone32-" + uuid4().hex[:16]
      status, job = call(base, f"/v1/deployments/{item}/checks?key={quote(key)}",
                         ctx, operator, "POST")
      if status != 202 or not job.get("id"):
         print("HOLD: task was not accepted")
         return 2
      deadline = time.monotonic() + 60
      while time.monotonic() < deadline:
         status, state = call(base, "/v1/checks/" + str(job["id"]), ctx, viewer)
         if status != 200:
            print("HOLD: cannot inspect task")
            return 2
         if state.get("status") in ("succeeded", "failed"):
            if state["status"] == "succeeded":
               print("PASS: staging worker completed durable job")
               return 0
            print("HOLD: staging worker returned a failed job")
            return 2
         time.sleep(2)
      print("HOLD: staging worker completion timeout")
      return 2
   finally:
      status, _ = call(base, f"/v1/deployments/{item}", ctx, operator, "DELETE")
      print("Staging fixture cleanup HTTP", status)
```

```
def main(argv=None):
    p = argparse.ArgumentParser()
    p.add_argument("--url", required=True)
    p.add_argument("--ca-file", default=None,
                   help="Trusted local CA PEM; omit for public HTTPS")
    p.add_argument("--private-dir", default="private")
    p.add_argument("--staging-write", action="store_true")
    a = p.parse_args(argv)
    try:
        return run(a.url, a.ca_file, a.private_dir, a.staging_write)
    except (OSError, ValueError, ssl.SSLError, URLError) as exc:
        print("HOLD: TLS/HTTP smoke error type " + type(exc).__name__, file=sys.stderr)
        return 2


if __name__ == "__main__":
    raise SystemExit(main())
```

Command 32.10 — HTTPS and operational inspection

```
# Local HTTPS; the CA PEM must be obtained from the trusted VM.
python3 ops32/smoke32.py --url https://localhost:8443   --ca-file ./approved-prod-ca.crt --private-dir private
# For publicly trusted HTTPS, omit --ca-file:
# python3 ops32/smoke32.py --url https://app.YOUR_DOMAIN
# ONLY against a disposable staging deployment:
# python3 ops32/smoke32.py --url https://STAGING_HOST #   --private-dir private/stage30 --staging-write
# Review bounded logs and runtime resource limits:
docker compose -p deployboard-prod28   -f compose.prod28.yaml -f compose.local28.yaml ps
docker stats --no-stream
# If jobs remain queued, verify publisher + worker separately from /ready.
```

Table 32.2 — Evidence expected after a real deployment

Observation	Expected proof	Release outcome
TLS	Valid verified chain; correct host; expected expiry	HOLD if any verification is bypassed
API	Health 200, ready 200, anonymous 401, viewer 200/403	HOLD if authentication differs
Database	Same deployment record after API recreation	HOLD if persistence fails
Worker	Staging job progresses to succeeded in bounded time	HOLD if queue remains stuck
Resources	CPU/memory policy active, errors absent	Investigate sustained saturation

Record Docker Engine and Compose versions, image digest, service container states, observed HTTPS status codes, request IDs, resource limits, and relevant timestamps. Keep logs bounded and redact sensitive material

before sharing. A running Caddy container is not proof of correct certificate trust, and a ready API is not proof that all background jobs have completed.

32.10 Demonstrate Backup, Restoration, and Rollback

Two recovery drills must be demonstrated separately. An application-image rollback replaces the API/publisher/worker image while retaining database state, and is safe only if schema changes are backward-compatible. A database restore can rewind business data and therefore requires a separately approved recovery point and data-loss decision. Chapter 29's PostgreSQL archive is intentionally restored to an isolated database, never casually onto the live volume.

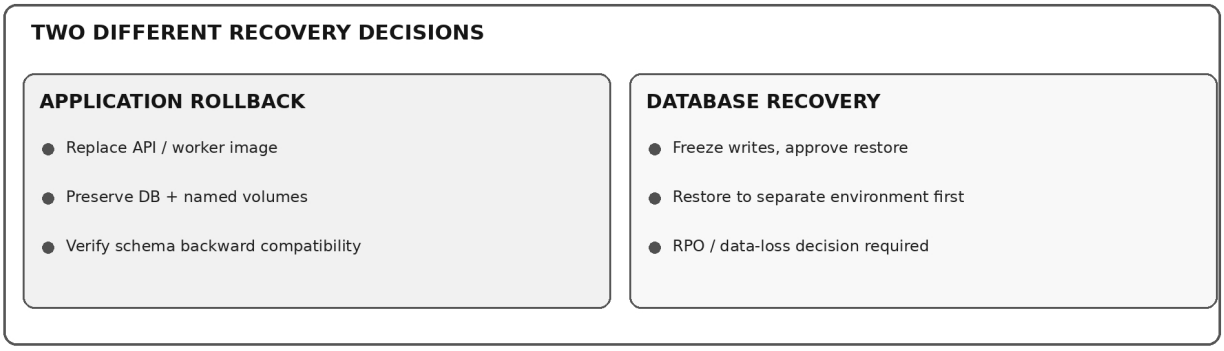


Figure 32.2 — Application rollback and database recovery have different scopes and approval requirements.

Command 32.11 — Create backup evidence and verify archive integrity

```
# On the approved production host with the digest env already set.
bash tools/backup29.sh ./backups29
# Find the newly created archive name in the output. Then substitute it:
python3 tools/archive29.py verify   ./backups29/DEPLOYBOARD_UTC_TIMESTAMP.dump
# Inspect the Chapter 29 recovery-only Compose file first.
docker compose -p deployboard-drill32   -f compose.recovery29.yaml config -q
# The actual isolated pg_restore and record-integrity queries follow
# Chapter 29 exactly. Capture elapsed time and verified row evidence.
# Never use production's project name or `down -v` in this drill.
```

Perform the Chapter 30 deliberately failed staging release with its dedicated fault override and staging secrets. Verify the previous image digest becomes active again while the staging PostgreSQL record remains intact. If a schema migration cannot be read by the previous binary, mark application-only rollback unsafe and use a forward fix or a formally approved data recovery. No rollback decision should be automated based on a single container-health indicator.

Command 32.12 — Create a conservative change/recovery decision record

```
python3 tools/release30.py plan   --previous 'REGISTRY/OWNER/deployboard@sha256:PREVIOUS_REAL_DIGEST'   --
candidate "$DEPLOYBOARD_IMAGE"   --backup ./backups29/DEPLOYBOARD_UTC_TIMESTAMP.dump   --mode local -
-out release30_capstone_plan.json
# The generated plan begins at HOLD and requires observed PASS evidence
# before the 'decide GO' action will accept a positive approval.
# Never substitute made-up digests or backup paths.
```

Measure restoration elapsed time against the recovery time objective (RTO). Compare the backup's creation time and any changes since it with the recovery point objective (RPO). If you have not rehearsed restoring a real valid dump in a separate project and compared record counts and relationships, claim neither RPO nor RTO success. For Redis, separately record any pending job effects and recovery actions: a PostgreSQL dump does not restore Redis's in-memory or AOF state.

32.11 Complete the Production Readiness Evidence Sheet

A signed release record should be the shortest trustworthy answer to five questions: what was shipped, what source and dependencies produced it, what tests were observed, what data can be recovered, and who authorized the result. The blank worksheet included with this chapter makes incomplete gates visible. Fill every row with a timestamped command result, location of stored evidence, and accountable owner. Do not mark a planned operation as PASS.

File 32.4 — ops32/EVIDENCE_SHEET.md (complete capstone acceptance sheet)

```
# DeployBoard Capstone — Production Readiness Evidence

Release ID: _____  UTC date: _____  Operator: _____
Environment: local / public  Project: deployboard-prod28

| Gate | Evidence location or observed output | Result (PASS/HOLD/PENDING) | Owner |
|---|---|---|---|
| Workspace & dependency inventory | | | |
| Pinned base and dependency versions | | | |
| Isolated QA tests and persistence | | | |
| Security audit, secrets and image scan | | | |
| Registry digest, platform, SBOM | | | |
| HTTPS certificate chain and expiry | | | |
| Unauthenticated denial + authenticated access | | | |
| Redis worker and job lifecycle | | | |
| Resource limits and service logs | | | |
| PostgreSQL backup archive + SHA256 | | | |
| Isolated restore data verification | | | |
| Failed-release rollback in staging | | | |
| RPO/RTO drill compared with objectives | | | |

Release decision: HOLD / APPROVED / ROLLED BACK
Reason and unresolved risks: _____
Approved recovery point (backup ID, location, SHA256): _____
Previous digest for application-only rollback: _____
```

```
Approval name/date: _____

A checksum records integrity against an expected value; it is not a signature.
Do not put passwords, token values, private keys, or raw sensitive logs in this sheet.
```

File 32.5 — tests/test_capstone32.py (complete offline utility checks)

```
"""Offline capstone checks; do not mistake these for a live release."""
import json
from pathlib import Path
import tempfile
import unittest
from unittest.mock import patch

from ops32.pin_base import render
from ops32.release_receipt import is_digest_image, receipt, matches, SOURCES
from ops32.smoke32 import safe_base, token_file

DIGEST = "ghcr.io/example/deployboard@sha256:" + "a" * 64
BASE = "python:3.13-slim@sha256:" + "b" * 64


class CapstoneChecks(unittest.TestCase):
    def test_pinned_base(self):
        result = render("FROM python:3.13-slim\nRUN echo hello\n", BASE)
        self.assertEqual(result.splitlines()[0], "FROM " + BASE)
        self.assertIn("RUN echo hello", result)

    def test_refuse_floating_base(self):
        with self.assertRaises(ValueError):
            render("FROM python:3.13-slim\n", "python:3.13-slim")

    def test_refuse_changed_source_dockerfile(self):
        with self.assertRaises(ValueError):
            render("FROM alpine:latest\n", BASE)

    def test_reject_tag_only_release(self):
        self.assertTrue(is_digest_image(DIGEST))
        self.assertFalse(is_digest_image("ghcr.io/example/deployboard:latest"))

    def test_source_tampering_detected(self):
        with tempfile.TemporaryDirectory() as work:
            root = Path(work)
            for rel in SOURCES:
                target = root / rel
                target.parent.mkdir(parents=True, exist_ok=True)
                target.write_text("safe-source", encoding="utf-8")
            record = receipt(root, DIGEST, "local")
            self.assertTrue(matches(root, record))
            (root / SOURCES[0]).write_text("changed", encoding="utf-8")
            self.assertFalse(matches(root, record))

    def test_https_only(self):
        self.assertEqual(safe_base("https://localhost:8443/"),
                         "https://localhost:8443")
        for bad in ("http://localhost", "https://u:p@example.com",
                    "https://host/path?token=leak"):
            with self.assertRaises(ValueError): safe_base(bad)

    def test_secret_file_refuses_symlink(self):
        with tempfile.TemporaryDirectory() as work:
            root = Path(work)
            target = root / "token.txt"
            target.write_text("value", encoding="utf-8")
            (root / "viewer_token.txt").symlink_to(target)
            with self.assertRaises(ValueError): token_file(root, "viewer_token")
```

```
    def test_receipt_does_not_record_secret_fields(self):
        with tempfile.TemporaryDirectory() as work:
            root = Path(work)
            for rel in SOURCES:
                target = root / rel
                target.parent.mkdir(parents=True, exist_ok=True)
                target.write_text("safe-source", encoding="utf-8")
            data = json.dumps(receipt(root, DIGEST, "public"))
            for field in ("viewer_token", "operator_token", "db_password"):
                self.assertNotIn(field, data)


if __name__ == "__main__":
    unittest.main()
```

Command 32.13 — Verify the chapter code, then approve manually

```
python3 -m unittest discover -s tests -p 'test_capstone32.py' -v
python3 ops32/release_receipt.py verify
# Complete ops32/EVIDENCE_SHEET.md with actual results.
# Sign only after real HTTPS, QA, registry, restore and rollback gates.
# Preserve a restricted, immutable copy outside the Docker host.
# Any PENDING or HOLD on a critical control means release approval = HOLD.
```

CAPSTONE PASS Your release is approved only if the clean-source inventory, actual pinned image digest, QA/persistence and security gates, verified HTTPS and access decisions, observed worker flow, independently tested recovery, rollback compatibility, and recorded operating owner all pass. Otherwise the result is HOLD with a documented next action—not an invented success.

32.12 Plan Further Learning: Swarm, Kubernetes, and Beyond

You have assembled a real delivery system rather than a collection of disconnected Docker commands. Its limits are equally real: a single Linux host is a failure domain; Compose startup dependencies do not provide continuous failover; local named volumes do not create distributed durable storage; and registry digests do not establish an end-to-end signing policy. Use measured requirements rather than trend pressure to decide what comes next.

Table 32.3 — Choose the next platform for a stated need

Next step	Explore when	Carry forward
Docker Swarm	You need a simpler built-in multi-node service model	Secrets, updates, service discovery, rollback practices
Kubernetes	You need scheduling, service controllers and mature policy ecosystem	Readiness, storage, resource requests/limits, signed artifacts
Managed data / cloud	You need outsourced failover and backups with clear SLAs	Restore verification, network trust and cost accounting
Stay with Compose	One host demonstrably meets your SLO, cost and skill limits	Everything in this capstone runbook

An operator who can answer “which digest, which database state, which credentials, which evidence, and which recovery route?” is already working at a professional standard. Keep the final evidence sheet, revisit it when dependencies change, and treat your next infrastructure decision as another testable engineering choice.

APPENDICES

Appendix A — Docker Installation and Recovery by Platform

Your Docker work depends on a healthy host installation. This appendix is a fault-isolation guide, not an invitation to reinstall whenever a command fails. Start with verification; preserve data; change one variable at a time; and stop as soon as the failing boundary is understood.

YOUR TARGET A working Linux-container Docker environment with a responding daemon, the Compose v2 plugin, an accessible image registry, and a successful disposable container run. On Linux, use sudo for Engine commands unless you have deliberately configured another secure access method.

Platform	Preferred learning installation	First boundary to check
Windows 10/11	Docker Desktop with WSL 2 backend	WSL version, VM virtualization, Linux-container mode
macOS	Docker Desktop for Apple silicon or Intel	Correct chip build, running Desktop backend
Ubuntu Linux	Docker Engine from official apt repository + Compose plugin	systemd daemon, socket permission, plugin

The universal check Run the native commands for your platform in this order. “Client” proves the executable exists; “Server” proves a daemon responds. A successful CLI version alone never proves Docker Engine is running.

POWERSHELL, TERMINAL, OR BASH (LINUX MAY REQUIRE SUDO)

```
docker --version
docker version
docker context ls
docker info
docker compose version
docker buildx version
docker run --rm hello-world
```

Expected proof: docker version includes both Client and Server; docker info identifies the expected server; docker compose version reports Compose v2; and hello-world downloads and prints its success message before exiting. If one test fails, use Figure A.1 and the relevant platform subsection rather than jumping to reset.

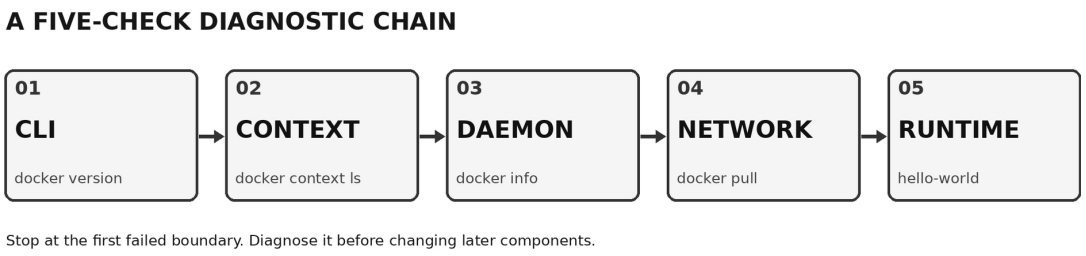


FIGURE A.1 Five checks isolate the failure boundary before recovery.

A.1 Windows with WSL 2: Verification and Common Fixes

Docker Desktop usually runs Linux containers through a WSL 2-backed Linux environment. You do not need to install Docker Engine independently inside each WSL distribution to use Docker Desktop. Keep the Desktop backend and the distribution integration as separate diagnostic questions.

Step 1 — verify WSL Open PowerShell. Run the following commands before changing any Windows features. WSL commands run on Windows, not inside an Ubuntu Bash prompt.

WINDOWS POWERSHELL

```
wsl --version
wsl --status
wsl --list --verbose
```

A supported WSL release must be installed; Docker currently documents WSL 2.1.5 or later for its WSL backend. If you plan to use Enhanced Container Isolation, consult the current Docker Desktop requirements rather than assuming an older WSL kernel is sufficient. Your Linux distribution should report version 2 when you intend to use it for integrated development.

Step 2 — repair an outdated or absent WSL installation In an elevated PowerShell window, use the minimum relevant command. A reboot may be required after enabling Windows features; update your WSL installation before retrying Docker Desktop.

WINDOWS POWERSHELL — USE ONLY AS NEEDED

```
wsl --update
# On a machine without WSL, in elevated PowerShell:
wsl --install
# For a specific distribution already installed as WSL 1:
wsl --set-version Ubuntu 2
```

IF VIRTUALIZATION FAILS Check Task Manager > Performance > CPU for “Virtualization: Enabled.” If firmware virtualization or Windows virtualization features are disabled, coordinate an approved change; a Docker reinstall cannot repair firmware configuration. If Windows requests a restart, finish it before retesting.

Step 3 — install and integrate Install the current Docker Desktop Windows build from Docker's official installer page. Select the WSL 2 backend if offered, launch Docker Desktop, then verify Linux containers are selected. If you work inside Ubuntu on WSL, enable that distribution under Docker Desktop Settings > Resources > WSL Integration where available.

WINDOWS POWERSHELL — AFTER DOCKER DESKTOP STARTS

```
docker version
docker context show
docker compose version
docker run --rm hello-world
```

For WSL-based development, store frequently accessed source trees inside the Linux filesystem (for example, ~/projects/deployboard) rather than a mounted Windows drive when performance and file-change notifications matter. Run docker from the integrated WSL shell only after confirming the Desktop integration.

Symptom	Differentiate	Targeted response
docker: command not found in Ubuntu	Works in PowerShell but not WSL shell	Enable WSL integration for that distro; restart the distro shell.
Cannot connect to daemon	docker --version works; docker version lacks Server	Start Desktop; check context and engine status, then retry.
Virtual machine fails at launch	wsl --status or virtualization check fails	Update WSL; enable prerequisites; reboot as prompted.
Bind mounts seem slow	Source under /mnt/c or other Windows-mounted path	Move active source to the WSL Linux filesystem.
Cannot pull image	Server responds but hello-world pull fails	Troubleshoot registry access in A.4, not WSL installation.

AVOID DUPLICATE DAEMONS When Docker Desktop supplies Engine access to WSL, installing a second daemon inside the same distro introduces confusing socket and context behavior. Use one deliberate configuration and record docker context show.

A.2 macOS: Architecture, Permissions, and Resource Settings

On macOS, Docker Desktop runs Linux containers inside a managed virtual machine. macOS does not run the Linux Docker daemon natively. The installed Desktop application must be running even if the docker command-line tool is already on your PATH.

Step 1 — identify your processor and prerequisites Choose the correct official installer: Apple silicon (arm64) or Intel (x86_64). The current Docker instructions require a supported macOS release and at least 4 GB of RAM; more memory may be useful for the full DeployBoard database-and-worker stack.

MACOS TERMINAL

```
uname -m
sw_vers -productVersion
command -v docker
```

```
docker --version
```

uname -m returns arm64 on Apple silicon and x86_64 on Intel. After opening the downloaded Docker.dmg, move Docker.app into Applications and start it. Accept the subscription agreement if presented. On some current configurations the CLI is installed under ~/.docker/bin; if the shell cannot find docker, inspect Docker Desktop Settings > Advanced and verify the PATH setup before reinstalling.

Step 2 — check the running backend and resources Open Docker Desktop, confirm it is running, then inspect Settings > Resources for available memory, CPU, disk image, and file-sharing controls. The exact settings presented vary by Desktop version and Mac virtualization backend; do not copy another machine's memory limit without checking your free host memory.

MACOS TERMINAL — AFTER DESKTOP STARTS

```
docker version
docker info
docker compose version
docker run --rm hello-world
```

If a Compose service exits under load, inspect docker stats and container exit status before increasing VM memory. If a bind mount fails, confirm that the host path exists and that macOS privacy or Docker file-sharing permissions allow access. For a cross-platform image mismatch, inspect the image's available architectures; an amd64-only image may run under emulation or fail on arm64.

MACOS TERMINAL — DIAGNOSTIC CHECKS

```
docker image inspect hello-world --format "{{.Os}}/{{.Architecture}}"
docker system df
docker context ls
```

PERMISSIONS AND EMULATION Do not disable macOS security checks or grant whole-disk access blindly to work around a single mount failure. Prefer a narrow, approved folder; prefer native arm64 images where available. Rosetta 2 is not universally required for ordinary arm64 container work.

Observed issue	Best first check	Safer action
CLI missing	command -v docker; Desktop CLI settings	Fix PATH or CLI tool placement.
Engine unavailable	docker version Server section; Desktop status	Start or restart Desktop; verify context.
Bind mount rejected	Host path, app permissions, file-sharing scope	Authorize only the required project folder.
Out-of-memory exit	docker inspect exit code; docker stats	Size resources from evidence; retry fixture workload.
Wrong image platform	uname -m; image manifest/architecture	Select matching platform/image; document emulation.

A.3 Linux: Engine, CLI, Compose Plugin, and Rootless Notes

A Linux server commonly installs Docker Engine directly. The tested reference path here is Ubuntu 24.04/26.04 LTS using Docker's official apt packages. On Debian,

Fedora, or another distribution, follow that distribution's official Docker Engine installation page rather than applying Ubuntu repository URLs unchanged.

Step 1 — inspect before installing Check the OS and determine whether Docker or containerd is already installed. Package conflicts must be resolved deliberately; the commands below assume a fresh Ubuntu installation rather than an already-managed production host.

UBUNTU BASH

```
cat /etc/os-release
command -v docker || true
systemctl status docker --no-pager || true
apt-cache policy docker-ce docker.io 2>/dev/null | head -30
```

Step 2 — configure the signed official repository Run each block in Ubuntu Bash, as a user allowed to use sudo. This is a repository setup, not a request to delete existing application state.

UBUNTU BASH — INSTALL PREREQUISITE AND SIGNING KEY

```
sudo apt update
sudo apt install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
  -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

UBUNTU BASH — CREATE THE DOCKER APT SOURCE

```
sudo tee /etc/apt/sources.list.d/docker.sources >/dev/null <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
Components: stable
Architectures: $(dpkg --print-architecture)
Signed-By: /etc/apt/keyrings/docker.asc
EOF
sudo apt update
```

Step 3 — install Engine and the supported plugins These are separate packages: Docker Engine, the CLI, the container runtime, Buildx, and Compose v2. If you already have the distribution's docker.io or legacy docker-compose package, examine package conflicts first rather than combining package families.

UBUNTU BASH — ENGINE INSTALLATION AND VERIFICATION

```
sudo apt install -y docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin
sudo systemctl enable --now docker
sudo docker version
sudo docker compose version
sudo docker run --rm hello-world
```

With default rootful Engine, using sudo docker is expected. The docker group gives its members root-equivalent authority through the daemon socket; do not add users automatically merely to eliminate sudo. Rootless mode runs both daemon and

containers without root privileges but needs uid/gid mappings and compatible networking, storage, and cgroup support. Consult Docker's rootless instructions for your host before switching an existing deployment.

When the Engine service fails Check logs and socket/context status. Treat an inactive service, an invalid /etc/docker/daemon.json, a permission-denied socket, and a failed image pull as distinct faults.

UBUNTU BASH — READ-ONLY DIAGNOSIS

```
sudo systemctl status docker --no-pager
sudo journalctl -u docker -n 60 --no-pager
docker context show
ls -l /var/run/docker.sock
sudo docker info
```

DO NOT FIX SOCKETS WITH CHMOD 666 World-writable Docker socket permissions grant dangerous host control. Fix the service, context, account authorization, or deliberate rootless configuration instead. Docker-published ports can also bypass assumptions about UFW filtering; review host and DOCKER-USER rules in your deployment plan.

A.4 Proxy, VPN, Corporate Network, and Registry Access Issues

A Docker pull can fail even when installation is correct. Split local Engine health from external registry connectivity. A laptop VPN, transparent proxy, corporate TLS inspection, DNS resolver, and registry sign-in can fail independently. Record the exact error class without sharing access tokens or internal proxy credentials.

Step 1 — locate the network boundary Run the platform-appropriate hostname lookup and HTTPS test, then compare with an actual image pull. The registry /v2/ endpoint commonly returns HTTP 401 when the TLS connection works but authentication is required; that result alone is not a network failure.

TERMINAL / BASH (POWERSHELL: USE NSLOOKUP AND CURL.EXE)

```
docker version
docker pull hello-world
# macOS/Linux:
nslookup registry-1.docker.io
curl -I https://registry-1.docker.io/v2/
```

Signal	Likely boundary	Next action
Could not resolve host	DNS, VPN, or resolver	Verify DNS while on/off approved VPN.
TLS certificate unknown	TLS inspection or CA trust	Get approved corporate root CA guidance; do not disable TLS.
Proxy authentication required (407)	Corporate proxy credentials/settings	Use supported proxy configuration and secure credential storage.
Registry unauthorized (401)	Auth missing or expired for protected resource	Confirm account, token scope, and registry address.
Rate limit or 429	Registry quota	Sign in appropriately or wait; avoid repeated loops.
Timeout while pulling	Firewall, proxy, VPN, or outage	Test alternate approved network and check incident status.

Step 2 — select the correct proxy control plane For Docker Desktop, use its Settings > Proxies interface. Docker's daemon.json proxy settings are not used by Docker Desktop. For a standalone Linux Engine, an approved daemon proxy may be configured in /etc/docker/daemon.json, but editing that file requires a planned daemon restart and validation of existing settings.

EXAMPLE ONLY — LINUX ENGINE DAEMON.JSON PROXY FRAGMENT

```
{
 "proxies": {
  "http-proxy": "http://proxy.example.test:3128",
  "https-proxy": "http://proxy.example.test:3128",
  "no-proxy": "localhost,127.0.0.1,.internal.example.test"
 }
}
```

The proxy addresses above are placeholders; do not use them unchanged. Do not overwrite an existing daemon.json without merging its current valid keys. Container/build proxy variables are a different layer from the daemon's registry pull connection; storing passwords in Dockerfile ENV or dumping environment variables into bug reports risks leaks.

Step 3 — isolate VPN effects Check whether VPN software changes DNS resolution, HTTPS certificate trust, split-tunnel rules, or routes to private registries. Test only on authorized networks. Use a vendor-approved bypass/allowlist or ask your administrator to configure registry endpoints; do not defeat company network controls.

OFFLINE FALLBACK If no registry can be reached, collect diagnostics and validate only local files, such as docker compose config --quiet. Image-dependent checks must remain pending; don't claim that a successful local YAML parse proves a container can start.

A.5 Version Checks, Diagnostics, and Clean Reinstallation

The safest recovery sequence is reversible: observe -> locate the failed boundary -> repair one component -> verify -> document the result. Restarting a backend may be appropriate; resetting Docker Desktop or deleting /var/lib/docker is not a routine troubleshooting step.

Record a portable version inventory Capture versions and selected context without copying sensitive docker info output into public forums. Use these read-only commands; on Linux use sudo where needed for server information.

CROSS-PLATFORM CLI DIAGNOSTIC INVENTORY

```
docker --version
docker version
docker compose version
docker buildx version
docker context ls
docker system df
```

```
docker ps -a
```

Collect platform evidence Docker Desktop: open Troubleshoot > Get support or run docker desktop diagnose if your installed version supports it. Linux Engine: inspect sudo journalctl -u docker and systemctl status docker. Check logs for tokens, proxy URLs, private hostnames, and filesystem paths before sharing.

DOCKER DESKTOP — SUPPORTED INSTALLATIONS ONLY

```
docker desktop diagnose
```

LINUX ENGINE — UBUNTU BASH

```
sudo systemctl status docker --no-pager
sudo journalctl -u docker -n 80 --no-pager
```

Before making a disruptive change, inventory deployed projects and named volumes; preserve the Compose files, credential sources, registry image references, and a verified logical database backup. Chapter 29 gives the PostgreSQL dump-and-restore method. A Docker Desktop factory reset or disk-image cleanup can destroy local container and image data; any reinstall must have a tested way to recover important records.

Escalation level	Action	Data risk / evidence
1 — Observe	Check versions, context, daemon, DNS, proxy	Read-only; save sanitized errors.
2 — Targeted fix	Repair PATH, WSL integration, proxy, permissions, or config	Retest failing boundary only.
3 — Restart	Restart Docker Desktop or systemd Docker service	May interrupt running applications.
4 — Repair install	Update/reinstall package or Desktop preserving supported data	Confirm backups and recovery plan first.
5 — Reset data	Factory reset, data cleanup, or removal of runtime data	Potentially destructive; last resort only.

Final verification — ready for DeployBoard A platform is ready for the book only after Client and Server respond; docker compose version succeeds; the correct context is active; hello-world runs; available storage is sufficient for PostgreSQL and builds; and required registries can be reached. On an offline host, mark the last two checks accordingly rather than assigning a false PASS.

LAST-MILE ACCEPTANCE COMMANDS

```
docker version
docker compose version
docker context show
docker run --rm hello-world
docker system df
```

RETURN TO THE PROJECTS Use the same trusted Docker context and project directory when you return to DeployBoard. If a named volume seems missing after reinstalling or switching contexts, first inspect docker context show and docker compose ls; do not create new empty data volumes until you understand where the existing records reside.

Installers, OS support windows, settings labels, and minimum requirements change. Confirm the current official page for your platform before reproducing system-level

commands; the commands here intentionally avoid pinned, aging product version numbers.

Appendix B — Docker Command-Line Quick Reference

Find the correct command. Confirm the target. Protect the data.

A field reference is useful only when it helps you choose the smallest safe command for the question you are trying to answer. Use the sequence observe → inspect → change → verify; do not begin with prune or force. Most examples run unchanged in PowerShell or Bash because they are Docker CLI commands. Copy one line at a time without a shell prompt.

> **CONVENTIONS** Replace uppercase placeholders such as ACCOUNT, USER, HOST, CONTAINER, IMAGE, and VOLUME before running commands. Example registry namespace example/deployboard-api is illustrative. The lab container db-cli-lab is optional; start it first if you want to run its inspection examples. For the complete DeployBoard stack, use the actual service names from docker compose config --services. Run Docker commands only against a daemon you are authorized to manage.

B.1 Container Lifecycle and Inspection Commands

Start with evidence. Read-only inspection commands come before restarts, container replacement, and cleanup. The current Docker context controls which Engine receives your command.

KNOW WHICH ENGINE YOU ARE TOUCHING

The same CLI command can act on a local Desktop Engine or a remote server

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker --version	Check the installed Docker CLI version; not a daemon-health check.
docker version	Compare client/server versions; a server connection error points toward the daemon or context.
docker info	Inspect Engine settings, storage driver, server OS, and active resources; output may reveal environment details.
docker context show	Confirm the Docker endpoint targeted by the next command; essential before remote changes.
docker ps	List running containers on the selected Engine.
docker ps -a	Show stopped, failed, and created containers too.
docker ps --filter status=exited	Narrow the list to containers that exited.
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"	Create a compact operational list of names, status, and published ports.

START, STOP, AND INSPECT CONTAINERS

Run db-cli-lab examples only after creating the optional lab container. They do not operate DeployBoard.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker run --rm hello-world	Pull and run a one-shot smoke test; the container is removed afterward.
docker run -d --name db-cli-lab -p 127.0.0.1:8088:80 nginx:stable-alpine	Start an optional local-only Nginx lab; the tag is for practice, not production pinning.
docker start db-cli-lab	Start an existing stopped lab container without recreating it.
docker stop db-cli-lab	Ask a running container to stop gracefully; expect downtime.
docker restart db-cli-lab	Restart the container; inspect logs if behavior does not recover.
docker inspect db-cli-lab	Read detailed configuration/state; environment and mount paths may be sensitive.
docker inspect --format '{{.State.Status}}' db-cli-lab	Extract a single state field, easier to record in runbooks.
docker inspect --format '{{.State.OOMKilled}}' db-cli-lab	Check whether the container was recorded as OOM-killed; also investigate other termination evidence.
docker port db-cli-lab	Show host-to-container port mappings.
docker exec -it db-cli-lab sh	Open an interactive shell in a running container that includes sh.
docker cp db-cli-lab:/etc/nginx/nginx.conf ./nginx.conf	Copy the sample Nginx configuration OUT of the container for offline inspection.
docker stats --no-stream	Capture one resource-usage sample instead of a continuous dashboard.

LOGS, PROCESSES, AND EVENTS

Prefer bounded log windows while investigating a specific incident.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker logs --tail 60 --timestamps db-cli-lab	Read recent timestamped output; may contain sensitive application data.
docker logs --since 15m db-cli-lab	Limit the log window to recent evidence.
docker logs -f --tail 20 db-cli-lab	Follow live logs; press Ctrl+C to stop following (container continues).
docker events --since 15m	Inspect Engine events such as container restarts and network changes.
docker top db-cli-lab	View processes running in the selected container.
docker diff db-cli-lab	See filesystem changes in the writable container layer, not named-volume changes.

DOCKER COMPOSE AT THE PROJECT LEVEL

Run these in a directory with the correct Compose file or supply explicit -f and -p flags.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker compose config --quiet	Validate merged Compose configuration without printing the resolved configuration.
docker compose config --services	List effective service names before using exec, logs, or up.
docker compose ps --all	Inspect all project services, including those that stopped.
docker compose up -d	Create or update services and return the terminal to you.
docker compose logs --tail 50 --timestamps api	Read recent logs for a service named api (adapt to your Compose file).
docker compose exec api sh	Enter a running API container if it includes a shell.
docker compose restart api	Restart an existing service container; it does NOT apply all changed configuration.
docker compose up -d --force-recreate api	Recreate a service to apply configuration changes; confirm impact first.
docker compose down	Remove this Compose project's containers and ordinary project networks; named volumes remain by default.

COMPOSE CAUTION docker compose config (without --quiet) prints resolved configuration, which may contain environment-variable values or sensitive paths. Avoid publishing raw output in bug reports.

B.2 Image Build, Tag, Pull, and Push Commands

Local builds and registry releases are different. A local image ID identifies locally stored image content; a repository digest identifies registry content. Tags are convenient labels and may move. Deployment records should retain a registry digest.

BUILD AND INSPECT AN IMAGE

The examples assume a Dockerfile is present in the current workspace.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker build -t deployboard-api:1.0.0 .	Build the current directory as context; requires a Dockerfile and its referenced files.
docker build -f Dockerfile -t deployboard-api:1.0.0 .	Explicitly select the Dockerfile and build context.
docker buildx build --check .	Check Dockerfile/build configuration without producing the final image.
docker buildx build --load -t deployboard-api:1.0.0 .	Use BuildKit/Buildx and load a single-platform result into the local image store.
docker buildx build --progress=plain --load -t deployboard-api:1.0.0 .	Show detailed, reproducible build output for troubleshooting.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker buildx build --pull --load -t deployboard-api:1.0.0 .	Ask BuildKit to check for newer base images; a mutable tag may change the build.
docker buildx build --no-cache --load -t deployboard-api:1.0.0 .	Rebuild without layer cache when testing a cache hypothesis; not a substitute for pinned inputs.
docker image ls	List locally available images and tags.
docker image inspect deployboard-api:1.0.0	Inspect an image; output may expose environment and image metadata.
docker image history deployboard-api:1.0.0	Review layers and build history for unexpected content.
docker image tag deployboard-api:1.0.0 example/deployboard-api:1.0.0	Add a repository-specific tag; does not create new image content.

PUBLISH, RETRIEVE, AND TRANSFER IMAGES

Replace example with your registry account or organization. Never commit an access token into source files.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker login --username ACCOUNT	Authenticate interactively; use a limited-scope token where the registry supports one.
docker push example/deployboard-api:1.0.0	Upload a tagged image to a registry you are authorized to write to.
docker pull example/deployboard-api:1.0.0	Pull the named tag; tags can be repointed by their publisher.
docker image inspect --format "{{json .RepoDigests}}" example/deployboard-api:1.0.0	Read known repository digests after push or pull; do not confuse the image ID with a registry digest.
docker buildx imagetools inspect example/deployboard-api:1.0.0	Inspect registry manifest information and advertised platforms.
docker pull example/deployboard-api@sha256:REPLACE_WITH_REAL_DIGEST	Pull immutable registry content AFTER substituting an actual 64-hex-character digest.
docker logout	Remove saved registry login credentials for the active registry endpoint.
docker save -o deployboard-api.tar deployboard-api:1.0.0	Export a locally stored image to a tar archive for controlled offline transfer.
docker load -i deployboard-api.tar	Import image archive content and its available tags into the local image store.

BUILDX, MULTI-PLATFORM IMAGES, AND EVIDENCE

Multi-platform publication requires a compatible builder, registry, and supported targets.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker buildx version	Confirm the Buildx component is installed.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker buildx ls	List builders and their drivers/platform capabilities.
docker buildx inspect --bootstrap	Inspect and initialize the selected builder when necessary.
docker buildx build --platform linux/amd64,linux/arm64 --push -t example/deployboard-api:1.0.0 .	Publish a multi-platform image if the builder and registry support the requested platforms.
docker buildx build --sbom=true --provenance=true --push -t example/deployboard-api:1.0.0 .	Publish with SBOM/provenance attestations where supported; inspect the registry result.

SECURITY REMINDER Do not put credentials in Dockerfile ARG or ENV. For temporary build credentials, use BuildKit --secret and RUN --mount=type=secret; scrub build logs and release artifacts.

B.3 Networking, Volumes, Contexts, and System Cleanup

Identify boundaries before changing them. A named volume is not the same as a container writable layer. A published port is not the same as an internal service port. A Docker context can silently point at a remote daemon.

NETWORKS AND PUBLISHED PORTS

Networks are local to the selected Engine unless you use an explicitly configured multi-host networking solution.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker network ls	Inventory Engine networks.
docker network inspect db-cli-net	See members and settings of a lab network created with the command below.
docker network create db-cli-net	Create an isolated user-defined bridge network for a disposable lab.
docker network connect db-cli-net db-cli-lab	Attach the lab container to the lab network.
docker network disconnect db-cli-net db-cli-lab	Detach it from the lab network; may disrupt connections.
docker network rm db-cli-net	Delete the lab network only after its member containers have disconnected.
docker compose ps	Inspect published ports for a running Compose project.

VOLUMES AND MOUNTS

Treat database and uploaded-file volumes as application data, not disposable cache.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker volume ls	List local Docker volumes; volume names alone do not tell you what is safe to delete.
docker volume create db-cli-scratch	Create a disposable named volume for experimentation.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker volume inspect db-cli-scratch	Inspect its mountpoint and metadata; does not read the stored application records.
docker volume rm db-cli-scratch	Delete the named scratch volume ONLY after confirming it contains no needed data.
docker inspect --format "{{json .Mounts}}" db-cli-lab	Inspect container bind mounts and named volumes; paths can be sensitive.
docker compose config --volumes	List effective Compose volume names; note resolved paths/config may contain sensitive information elsewhere.

LOCAL AND REMOTE DOCKER CONTEXTS

Use the per-command --context flag for one-off remote inspection; avoid accidental sticky switching.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker context ls	List known Docker endpoints; * marks the active context.
docker context show	Name the active endpoint before every potentially destructive command.
docker context inspect default	Inspect the default endpoint definition.
docker --context default ps	Target one context for a single read-only command without switching defaults.
docker context create lab-remote --docker host=ssh://USER@HOST	Define an SSH-backed remote context after SSH access and host identity are verified.
docker --context lab-remote ps	Read container status on the remote host without changing the persistent default context.
docker context use default	Return to the local default when you intentionally changed contexts.

DISK USE AND SCOPED CLEANUP

Read docker system df before deciding which resource class you can safely reclaim.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker system df	Summarize images, containers, volumes and build-cache space on this Engine.
docker system df -v	Inspect detailed disk usage before cleanup.
docker compose -p deployboard ps --all	Scope a query to a named Compose project; use the same project identity across operations.
docker compose -p deployboard config --quiet	Validate a named project without exposing rendered values.
docker builder prune	Interactively remove unused build cache; know that future builds may be slower.

COMMAND / OPTION	PURPOSE & INTERPRETATION
docker image prune	Interactively remove dangling image objects; do not treat it as a universal safe cleanup.
docker system prune	Interactively remove multiple classes of unused resources; carefully read the printed warning.

SCOPE MATTERS docker compose -p NAME commands affect that Compose project on the ACTIVE Docker context. Before operating on a remote host, confirm both the project name and docker context show.

B.4 Commands That Can Delete Data: Safety Warnings

Deletion is not troubleshooting. Do not use force-removal or broad prune commands while you are still trying to understand an outage. First list the targets, establish recoverability, and confirm you are using the correct Engine.

DESTRUCTIVE-COMMAND HAZARD MAP

Shown for recognition and risk assessment, not as a recommended cleanup sequence

COMMAND / OPTION	IMPACT / SAFETY CONDITION
docker rm -f CONTAINER	**HIGH** Force-removes a container and its writable layer; stop workloads and confirm persistent data is external.
docker image rm IMAGE	**MEDIUM** Removes a local image reference/content when possible; can remove an offline rollback option.
docker network prune	**MEDIUM** Removes unused networks; verify you selected the intended Engine/context.
docker volume rm VOLUME	**CRITICAL** Deletes the named volume's stored data. Never use on DeployBoard database volumes without a proven recovery plan.
docker volume prune	**HIGH** Deletes unused ANONYMOUS local volumes by default; unreferenced does not mean unimportant.
docker volume prune --all	**CRITICAL** Deletes unused named AND anonymous volumes; includes detached databases.
docker compose down --volumes	**CRITICAL** Removes Compose-declared named volumes and attached anonymous volumes; default down does not.
docker system prune --all	**HIGH** Removes stopped containers and all unused images, networks and build cache.
docker system prune --volumes	**HIGH** Also prunes eligible anonymous volumes in addition to ordinary system prune targets.
docker builder prune --all	**MEDIUM** Removes all unused build cache, which may cause a slow future rebuild.

DOCKER VOLUME DETAIL docker volume prune removes unused anonymous volumes by default; adding --all includes unused named volumes. docker system prune --volumes can remove eligible anonymous volumes. docker compose down --volumes removes declared named volumes and attached anonymous volumes; external volumes are not removed by Compose down.

The five-step deletion approval gate

1 Identify the target. Run docker context show, docker ps -a, docker volume ls, and docker compose ps --all. Record exact names.

2 Distinguish state. Ask whether data lives in a named volume, an anonymous volume, a bind mount, or a container writable layer.

3 Prove recoverability. For DeployBoard, require a current PostgreSQL backup AND a successful isolated restore test before risky storage changes.

4 Choose the narrowest action. Stop or recreate one known service instead of cleaning up the entire Engine; avoid -f / --force by default.

5 Verify after the action. Recheck service health, persistent records, volume inventory, and the deployment manifest. Record the result.

DEPLOYBOARD RULE Never delete a database volume to "fix" a startup error. First diagnose health, credentials, migrations, permissions, or storage pressure. If you must restore data, follow the isolated recovery procedure in Chapter 29.

Appendix C — Dockerfile, BuildKit, and Buildx Reference

Use this appendix when you create or review Dockerfiles, debug an unexpectedly slow build, move build steps into stages, or prepare an architecture-specific or multi-platform release. The examples form a standalone training lab: you can create every file shown here, without a GitHub repository or credentials. They do not modify DeployBoard's database or deployment volumes.

LAB CONVENTIONS Work in a fresh appendix-c-lab directory. Commands use Docker's Linux-container engine. Replace uppercase placeholders only where explicitly indicated. A builder must be available for build commands; registry pushes require a registry account and permission.

C.1 Common Dockerfile Instructions and Their Purposes

A Dockerfile describes a build, not a running service. Build instructions construct filesystem layers and image configuration; runtime settings such as port publishing, secrets, and persistent volumes belong in the deployment specification. The table is a selection guide, not an instruction to use every directive.

Instruction	What it controls	DeployBoard decision
FROM	Starting image and optional build-stage name	Choose a maintained Python image; pin its digest for approved releases.
ARG	Build-time parameter; not a credential vault	Use for non-secret build options or versions.
ENV	Image environment default persisted at runtime	Safe for non-secret Python flags; never put tokens here.
WORKDIR	Working directory for later instructions	Use an explicit /service or /app path.
COPY	Context files or files from a build stage	Prefer for ordinary local source files.
ADD	Specialized copying, URLs, or archive handling	Use only when its extra behavior is needed.
RUN	Command executed while constructing an image	Install dependencies and perform build checks.
USER	Default runtime user	Use an unprivileged identity where feasible.
EXPOSE	Documentation of an expected listening port	Does not open or publish a host port.
CMD / ENTRYPOINT	Default container command / executable	Prefer exec-form arguments for signal handling.
HEALTHCHECK	Optional image-level health instruction	Keep readiness distinct from dependency availability.
LABEL / STOPSIGNAL	Metadata / stop signal	Label release metadata; set signal only when needed.

A COMPLETE MINIMAL RUNNABLE EXAMPLE

Create Dockerfile and hello.py with the exact contents below. They use only the Python standard library and serve one observable /health endpoint. The unprivileged user cannot alter the application file.

DOCKERFILE

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim
WORKDIR /app
RUN groupadd --system app && \
    useradd --system --gid app --no-create-home app
COPY --chown=app:app hello.py ./hello.py
USER app
EXPOSE 8080
CMD ["python", "hello.py"]
```

HELLO.PY

```
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
```

```
        if self.path != "/health":
            self.send_error(404)
            return
        body = b'{"status":"ok","service":"appendix-c"}'
        self.send_response(200)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)

if __name__ == "__main__":
    ThreadingHTTPServer(("0.0.0.0", 8080), Handler).serve_forever()
```

CHECK YOUR EXPECTATION EXPOSE 8080 does not publish a host port. The later docker run command binds the service only to 127.0.0.1:18080. A successful response is {"status":"ok","service":"appendix-c"}.

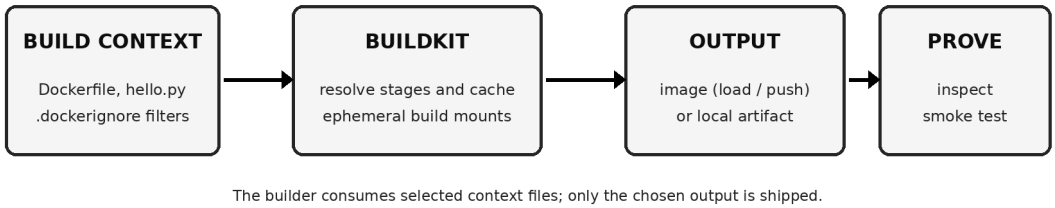


FIGURE C.1 Context filtering, BuildKit execution, selected output, and release verification.

C.2 Build Context, .dockerignore, and Caching Rules

The final dot in docker buildx build ... . selects the build context. COPY can access files from that context (subject to ignore rules), not arbitrary files elsewhere on your host. With BuildKit, required files may be transferred incrementally; this does not make sending an unnecessarily broad context a good practice. Keep backups, credentials, .git history, and generated artifacts out of the context.

.DOCKERIGNORE

```
# Ignore local state and credentials; do not exclude hello.py.
.git/
.env
.env.*
!.env.example
private/
*.dump
*.db
__pycache__/
*.pyc
.pytest_cache/
artifacts/
*.log
```

IGNORE-FILE PRECEDENCE A Dockerfile-specific ignore file named Dockerfile.NAME.dockerignore can take precedence for that Dockerfile. Do not assume the root .dockerignore applies identically to special test or

release Dockerfiles. Verify each context in its own build.

WHAT INVALIDATES A CACHE LAYER?

Change	Typical effect	Action
Edit hello.py	Source COPY and following steps change	Place dependency installation before frequent source copies.
Edit requirements.txt	Dependency-install layer must be refreshed	Use pinned dependencies and review actual upgrades.
Edit .dockerignore	Available build-context files change	Inspect generated context; never ignore required sources.
Edit a build secret value	Does not automatically invalidate cached RUN output	Use an explicit non-secret cache key when re-execution is required.
Cache mount purged	Build should remain correct; downloads may repeat	Cache improves speed, never correctness.

Build the example with visible step output; then repeat the same build. A second build often reuses steps, but elapsed time and cache hit rate depend on the active builder, base-image availability, and changed inputs.

BASIC BUILD AND REPEAT

```
docker buildx version
docker buildx build --progress=plain --load -t appendix-c:basic -f Dockerfile .
docker buildx build --progress=plain --load -t appendix-c:basic -f Dockerfile .
```

DEPENDENCY CACHING FOR THE MAIN DEPLOYBOARD IMAGE

The larger DeployBoard application already has a pinned requirements.txt. Its dependency installation can be separated from later source copies. If you want a BuildKit pip cache mount, do not combine it with --no-cache-dir: that pip option disables the package-download cache you are trying to reuse. The following is an adaptation pattern for that existing project, not a second runnable lab Dockerfile.

BUILDKIT PIP CACHE PATTERN (INSIDE A COMPLETE DOCKERFILE)

```
# syntax=docker/dockerfile:1
WORKDIR /service
COPY requirements.txt ./
RUN --mount=type=cache,target=/root/.cache/pip \
    python -m pip install -r requirements.txt
COPY app ./app
```

TRUST BOUNDARY Keep .env, private/, database dumps, and the Docker socket out of the build context. Passing an image through a cache is not the same as establishing its provenance or its security.

C.3 Multi-Stage, Secret-Mount, and Cache-Mount Patterns

Build stages let you compile or validate in one environment and transfer only chosen outputs into the runtime image. Name stages so that COPY --from=... is stable when stages are reordered. A smaller image is desirable, but preserving required shared libraries, certificates, and runtime behavior takes priority over size alone.

DOCKERFILE.MULTISTAGE — COMPLETE

```
# syntax=docker/dockerfile:1
FROM python:3.13-slim AS validation
WORKDIR /src
COPY hello.py .
RUN python -m py_compile hello.py

FROM python:3.13-slim AS runtime
WORKDIR /app
RUN groupadd --system app && \
    useradd --system --gid app --no-create-home app
COPY --from=validation --chown=app:app /src/hello.py ./hello.py
USER app
EXPOSE 8080
CMD ["python", "hello.py"]
```

VALIDATE A STAGE; LOAD THE FINAL IMAGE

```
docker buildx build --target validation -f Dockerfile.multistage .
docker buildx build --load -f Dockerfile.multistage -t appendix-c:1.0 .
```

The validation stage executes python -m py_compile; the runtime stage copies only hello.py. Neither Python base image is pinned by digest in this training lab. Before production, select an approved digest and verify the resulting image by a registry digest as described in Chapters 20, 25, and 32.

A SECRET THAT NEVER BECOMES AN IMAGE LAYER

BuildKit secret mounts exist only during the RUN step that requests them. They are safer than passing tokens through ARG, ENV, or COPY, but a careless command can still write a secret into a layer or build log. Never echo the value, upload it as an artifact, or include it in an image.

DOCKERFILE.SECRET — COMPLETE

```
# syntax=docker/dockerfile:1
FROM alpine:3.20
RUN --mount=type=secret,id=training_token,required=true \
    test -s /run/secrets/training_token && \
    printf 'Training credential supplied (value not printed)\n'
CMD ["sh", "-c", "printf 'Secret not persisted in image\n'"]
```

Create the folder private and a file named training-token.txt containing only a made-up training string. The root .dockerignore excludes private/. Supply the file explicitly as a build secret, then remove it after your exercise. The lab package deliberately contains no token file.

LINUX SHELL (JOIN THE LINES FOR POWERSHELL)

```
docker buildx build -f Dockerfile.secret --load \
  --secret id=training_token,src=private/training-token.txt \
  -t appendix-c:secret-check .
docker run --rm appendix-c:secret-check
```

EXPECTED PROOF A valid nonempty training secret lets the build pass without displaying its contents. The runtime command reports that no secret was saved. This is a demonstration, not a complete image leak audit.

CACHE MOUNTS ARE DISPOSABLE OPTIMIZATIONS

A BuildKit cache mount supplies a writable directory during RUN. It can save repeated downloads even if the layer must be rebuilt, but its contents are not guaranteed to survive builder cleanup or a fresh CI runner. Keep the build reproducible when the cache is empty. For package managers requiring exclusive access, such as apt, use sharing=locked.

REUSABLE PATTERNS; EACH RUN BELONGS IN A SUITABLE FULL DOCKERFILE

```
RUN --mount=type=cache,target=/root/.cache/pip \
    python -m pip install -r requirements.txt
# For apt only, prefer explicit locking:
RUN --mount=type=cache,target=/var/cache/apt,sharing=locked \
    --mount=type=cache,target=/var/lib/apt,sharing=locked \
    apt-get update && apt-get install -y ca-certificates
```

C.4 Build Checks, Multi-Platform Builds, and Artifact Outputs

Use build checks to catch suspicious Dockerfile patterns before building. Checks are available with compatible Buildx and Dockerfile frontends; they are not substitutes for unit tests, vulnerability scans, secret scans, or runtime health checks. A warning-free Dockerfile can still contain insecure application code.

PREFLIGHT AND SINGLE-PLATFORM LOCAL IMAGE

```
docker buildx ls
docker buildx inspect --bootstrap
docker buildx build --check -f Dockerfile.multistage .
docker buildx build --progress=plain --load \
  -f Dockerfile.multistage -t appendix-c:1.0 .
```

Run the sample on your local Engine. The name appendix-c-lab should not already exist. Use the provided Python verifier for a portable health check; it does not require curl or a shell inside the container.

PORTABLE ACCEPTANCE TEST (JOIN WRAPPED DOCKER LINES IN POWERSHELL)

```
python verify_lab.py
docker run --rm -d --name appendix-c-lab \
  -p 127.0.0.1:18080:8080 appendix-c:1.0
python verify_lab.py --url http://127.0.0.1:18080/health
docker stop appendix-c-lab
```

EXPECTED RESULT The verifier prints PASS for the six source files and PASS for HTTP 200 and a healthy JSON payload. A failed health check is a failure to investigate, not an acceptance result.

WHEN ONE IMAGE MUST RUN ON TWO CPU ARCHITECTURES

Objective	Command pattern / decision
One target for a local test	--platform linux/amd64 --load
Two architectures for a registry release	--platform linux/amd64,linux/arm64 --push
Inspect a manifest index by tag	docker buildx imagetools inspect REGISTRY/OWNER/IMAGE:VERSION
Immutable release reference	Deploy REGISTRY/OWNER/IMAGE@sha256:REAL_DIGEST

Buildx requires a builder capable of the requested targets. Emulation may be slower than native builders. A docker-container builder can push a multi-platform index directly to a registry but may not automatically load it into the local Engine. Do not claim that both architectures work until each has been tested.

OPTIONAL: AUTHENTICATED REGISTRY AND VERIFIED BUILDER REQUIRED

```
docker buildx build --platform linux/amd64,linux/arm64 \
  -f Dockerfile.multistage \
  -t REGISTRY/OWNER/appendix-c:1.0 --push .
docker buildx imagetools inspect REGISTRY/OWNER/appendix-c:1.0
```

EXPORT A FILESYSTEM ARTIFACT RATHER THAN AN IMAGE

The local exporter writes the root filesystem of the chosen final stage to a directory. This pattern can produce generated configuration, compiled executables, or an evidence file without publishing a container image. The following tiny Dockerfile builds a single README.txt artifact.

DOCKERFILE.ARTIFACT — COMPLETE

```
# syntax=docker/dockerfile:1
FROM alpine:3.20 AS write-file
RUN mkdir /out && \
    printf 'Appendix C artifact export - training example\n' > /out/README.txt
FROM scratch AS artifact
COPY --from=write-file /out/ /
```

ARTIFACT EXPORT AND SOURCE VERIFICATION

```
docker buildx build -f Dockerfile.artifact \
  --output type=local,dest=./artifacts .
python verify_lab.py
```

EXPECTED OUTPUT The file artifacts/README.txt should contain “Appendix C artifact export - training example”. The artifacts/ directory is ignored by the build context in this lab.

QUICK FAILURE DECODER

Evidence	Likely cause	First safe move
COPY failed: file not found	Wrong context path or ignore rule	Check .dockerignore and build location.
RUN --mount secret: missing	Unspecified secret ID or missing file	Confirm secret path locally; do not print its value.
no output specified warning	Builder uses a cache-only default	Choose --load, --push, or --output deliberately.
exec format error	Wrong target architecture or binary	Inspect platform and rebuild for the target.
Cached output appears stale	Input not tracked or cache key unchanged	Recheck COPY order and inputs; rebuild deliberately.

APPENDIX C ACCEPTANCE RECORD

Check	Evidence to retain
Files and ignore patterns	verify_lab.py prints PASS
Dockerfile checks	Buildx --check output and version
Runtime behavior	HTTP 200 from 127.0.0.1:18080/health
Secret handling	Nonempty dummy secret accepted; no value printed
Artifacts	artifacts/README.txt produced and inspected
Multi-platform release (optional)	Registry manifest and platform tests; no fabricated digest

APPENDIX D — Compose Configuration Reference and Patterns

A deployment-file field guide for the DeployBoard project: make intent visible, prove the resolved configuration, then change it safely.

A Compose file is not executable documentation until you can explain what Docker will create and why. This appendix uses one deliberately small DeployBoard fixture to show the boundaries between services, images, build contexts, data, networks, readiness, environment variables, and file-mounted credentials. Every example is printed in full and included as convenience files in the optional ZIP; no GitHub repository is required.

> ***SCOPE AND COMPATIBILITY*** *This reference targets modern Docker Compose V2 and Linux containers on Docker Desktop or Docker Engine. Run docker compose version before relying on an option. The examples were statically inspected and the Python HTTP fixture was tested offline; live Docker container execution remains a reader verification step.*

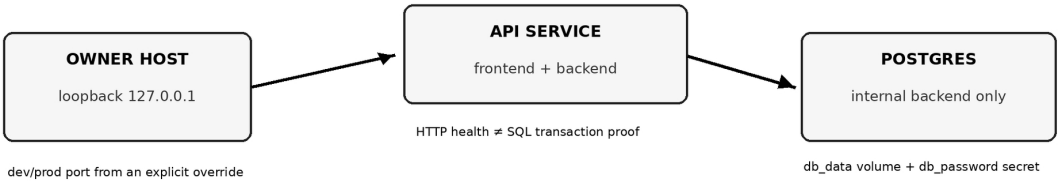


Figure D.1 The development or production override publishes the API only on host loopback. PostgreSQL stays on the private backend network; the API health endpoint does not certify SQL transaction readiness.

You want to prove	Command or artifact	Evidence to keep
Parse and merge configuration	docker compose ... config --quiet	Exit code and any diagnostic
Inspect interpolation	docker compose ... config --environment	Expected values; redact before sharing
Start and observe	docker compose ... up -d --build --wait	ps output and /health response
Inspect permissions and data	compose.yaml + inspect/volume checks	No database host port; named data volume
Change without surprises	explicit dev or prod -f override	Resolved config diff before up

D.1 Services, Images, Builds, Networks, and Volumes

Compose models an application rather than a list of shell commands. Each service describes one independently managed container role. Top-level networks and volumes declare resources reused by those roles; top-level secrets declare files eligible for explicit service grants. Use a named Compose project to avoid collisions with another DeployBoard installation.

Field	What it controls	DeployBoard decision
services.api	An app container built from local source	Build from ./api; no database password grant
services.db	An upstream PostgreSQL image	Use PostgreSQL 17 for this reference fixture
build versus image	Build context vs output image/tag	Use both; explicitly run up --build
networks	Container-to-container reachability	DB on internal backend only
ports	Host publication of container ports	Only in the chosen override, loopback bound
volumes	Lifecycle-independent database data	db_data persists across down; not down -v
secrets	File mounted for granted services	DB only; never turn passwords into image ENV

A service attached to backend can resolve db by service name. Docker's internal network is not an authorization system: the API has access to both frontend and backend, so it can initiate outbound traffic through its frontend attachment. Use

application authentication and egress controls where required. The API is purposely read-only and demonstrates Compose wiring; it does not issue SQL queries.

> ***IMAGE CHOICE*** *Postgres 17-alpine and python:3.12-slim are readable teaching tags, not immutable digests. For a pinned production rollout, resolve and record supported image digests and test planned patch upgrades. Do not treat a version tag alone as immutable.*

Minimum file map

Path	Purpose
compose.yaml	Canonical two-service model: readiness, networks, limits, volume, secret
compose.dev.yaml / compose.prod.yaml	Separate port and runtime policies; do not combine them
.env.example / api.env	Compose interpolation defaults / container environment
api/Dockerfile / api/main.py	Self-contained local HTTP demonstration
secrets/db_password.txt	Explicitly synthetic teaching value; replace securely before real use
verify_reference.py	Offline fixture contract; not a Compose parser

Complete reference model — compose.yaml

```
name: deployboard-reference

services:
  api:
    build:
      context: ./api
      dockerfile: Dockerfile
    image: deployboard-api:reference
    env_file:
      - ./api.env
    environment:
      APP_LABEL: ${APP_LABEL:-DeployBoard}
      APP_MODE: ${APP_MODE:-reference}
      DB_HOST: db
      DB_PORT: "5432"
    depends_on:
      db:
        condition: service_healthy
        restart: true
    healthcheck:
      test:
        - CMD
        - python
        - -c
        - "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8080/health', timeout=2)"
      interval: 10s
      timeout: 3s
      retries: 5
      start_period: 10s
```

```
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: "0.50"
          memory: 256M
    networks:
      - frontend
      - backend

  db:
    image: postgres:17-alpine
    environment:
      POSTGRES_DB: deployboard
      POSTGRES_USER: deployboard
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test:
        - CMD-SHELL
        - 'pg_isready -U "$${POSTGRES_USER}" -d "$${POSTGRES_DB}"'
      interval: 10s
      timeout: 3s
      retries: 8
      start_period: 30s
    restart: unless-stopped
    deploy:
      resources:
        limits:
          cpus: "1.00"
          memory: 512M
    networks:
      - backend

networks:
  frontend: {}
  backend:
    internal: true

volumes:
  db_data: {}

secrets:
  db_password:
    file: ./secrets/db_password.txt
```

Listing: compose.yaml (77 lines; copy without page headers or captions).

Minimal runnable API image — api/Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
```

```
COPY main.py /app/main.py
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
USER 10001:10001
EXPOSE 8080
CMD ["python", "/app/main.py"]
```

Listing: api/Dockerfile (7 lines; copy without page headers or captions).

Dependency-free demonstration server — api/main.py

```
"""Tiny read-only DeployBoard reference API (stdlib only)."""
import json
import os
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer


class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == "/health":
            status, data = 200, {"status": "ok", "service": "deployboard-api"}
        elif self.path == "/":
            status, data = 200, {
                "service": "deployboard-api",
                "label": os.getenv("APP_LABEL", "DeployBoard"),
                "mode": os.getenv("APP_MODE", "reference"),
                "database_host_configured": bool(os.getenv("DB_HOST")),
            }
        else:
            status, data = 404, {"error": "not-found"}
        body = json.dumps(data, sort_keys=True).encode("utf-8")
        self.send_response(status)
        self.send_header("Content-Type", "application/json; charset=utf-8")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)


if __name__ == "__main__":
    port = int(os.getenv("PORT", "8080"))
    ThreadingHTTPServer(("0.0.0.0", port), Handler).serve_forever()
```

Listing: api/main.py (30 lines; copy without page headers or captions).

D.2 Health Checks, depends_on, Startup Limits, and Restart Policies

A container is not ready merely because its process has started. The PostgreSQL healthcheck executes pg_isready inside the DB container. The API depends_on gate waits for db to be marked healthy before Compose starts the API. The API /health endpoint only demonstrates that its HTTP server responds; it does not prove a successful database login, schema migration, or business transaction.

Control	Exact meaning	Potential false assumption

Control	Exact meaning	Potential false assumption
healthcheck.test	Exit 0 means the specific check passed	“Healthy” covers every dependency
start_period	Grace period for early probe failures	Application must start before it expires
interval / timeout / retries	Probe cadence, time budget, failure threshold	One failed probe forces restart
depends_on.service_healthy	Wait for dependency health at Compose start	Automatically retries every failed app request
depends_on.restart: true	Propagate explicit Compose dependency restart/update	Automatically restarts after any DB crash
service restart: unless-stopped	Container exit/reboot restart policy	Automatically restarts an unhealthy process
deploy.resources.limits	CPU and memory ceilings where supported	Memory exhaustion cannot cause process exit

The double dollar sign in pg_isready ($${POSTGRES_USER}) intentionally defers substitution until the shell inside the PostgreSQL container evaluates the variable. A single dollar sign would invite host-side Compose interpolation before container startup. Healthcheck start_interval is an additional supported knob in newer Compose versions (introduced in Compose 2.20.2), omitted here for broader portability.

The limits in this reference set API to 0.50 CPU/256M and database to 1.00 CPU/512M. These are laboratory ceilings, not universal production sizes. Confirm local enforcement and observed resource use with docker inspect and docker stats; test heavier workloads before adoption.

> ***RESTART DISTINCTION*** *Service-level restart: unless-stopped deals with process exit. depends_on: {db: {restart: true}} coordinates explicit Compose restart/update operations. Neither setting repairs an application that is alive but incorrect. Your health endpoint and alerting must supply that missing evidence.*

At the keyboard

```
docker compose version
docker compose -f compose.yaml -f compose.dev.yaml config --quiet
docker compose -f compose.yaml -f compose.dev.yaml up -d --build --wait
docker compose -f compose.yaml -f compose.dev.yaml ps
docker compose -f compose.yaml -f compose.dev.yaml logs --tail=60 db
docker compose -f compose.yaml -f compose.dev.yaml logs --tail=60 api
```

Expected proof: db and api are reported healthy after startup; http://127.0.0.1:8088/health responds with JSON status ok. If readiness does not appear, inspect the healthcheck command inside its target image and confirm the database has the expected user and name. Do not interpret “up completed” as proof of end-to-end database access.

Controlled break-and-repair decision

BREAK: in a disposable copy of this fixture, temporarily replace the db healthcheck test with ["CMD-SHELL", "exit 1"]. VERIFY: run config --quiet and then up --wait in

that disposable project; configuration parsing should pass, but startup health should fail or time out. REPAIR: restore the original pg_isready probe and rerun up --wait. SHIP: record the actual resolved model and observed health result. Do not perform this fault exercise against a live DeployBoard database.

D.3 Environment Interpolation, env_file, and Precedence

Two different mechanisms share the phrase “environment file.” Compose interpolation substitutes variables into the YAML model before containers are created. The service-level env_file key injects values into the container environment. A value in the shell or .env does not enter the container unless the Compose model passes it through.

Copy this file to .env — interpolation defaults

```
# Local demonstration settings, not credentials.
APP_LABEL=DeployBoard Demo
APP_MODE=reference
HOST_HTTP_PORT=8088
PROD_HTTP_PORT=8090
```

Listing: .env.example (5 lines; copy without page headers or captions).

Values loaded by the service-level env_file

```
# Passed to the API container by env_file.
APP_LABEL=env_file_label
APP_MODE=env_file_mode
TRACE_LEVEL=info
```

Listing: api.env (4 lines; copy without page headers or captions).

Question	Where to check	Expected result
Where does ${APP_LABEL} come from?	Shell, then --env-file or .env	Shell APP_LABEL wins for substitution
What if APP_LABEL is unset?	${APP_LABEL:-DeployBoard}	Built-in default DeployBoard
What enters the API?	Resolved environment plus env_file	environment.APP_LABEL overrides api.env label
What does env_file alone do?	TRACE_LEVEL in api.env	TRACE_LEVEL=info in API
Can an env_file override environment?	environment wins for the same key	No, when explicitly mapped
Can an image ENV be overridden?	Service env_file/environment	Yes; higher precedence

The complete runtime precedence for the same container variable is: (1) docker compose run -e override; (2) interpolated environment/env_file value; (3) literal

service environment; (4) service env_file; (5) image ENV. That is distinct from interpolation input precedence: host shell, then explicitly passed --env-file, then the default project .env when no explicit file is provided. When multiple --env-file arguments are supplied, later files can override earlier values.

At the keyboard

```
cp .env.example .env  # macOS / Linux / WSL2
# Windows PowerShell: Copy-Item .env.example .env
docker compose -f compose.yaml -f compose.dev.yaml config --quiet
docker compose -f compose.yaml -f compose.dev.yaml config --environment
docker compose -f compose.yaml -f compose.dev.yaml config --services
```

> ***DO NOT LEAK SECRETS*** *docker compose config and config --environment may expose resolved environment values. This example uses no real credentials in interpolation. Keep production secrets out of .env, command history, screenshots, logs, and publicly shared configuration exports.*

PowerShell inspection example: set $env:APP_LABEL = "FromShell", run docker compose ... config, then Remove-Item Env:APP_LABEL. A blank variable may differ from an unset variable with ${VAR-default} versus ${VAR:-default}. Use ${VAR:?message} to reject missing required production values before creating containers.

D.4 Compose Overrides, File-Mounted Secrets, and Validation Commands

Overrides should make small, inspectable changes to a valid base model. Because Compose merges service mappings by key but handles some lists with uniqueness rules, reusing a port mapping with a different host port can append an additional published port. This fixture deliberately defines no host ports in compose.yaml; only one explicitly chosen override adds loopback publication. Do not pass both dev and prod overrides together.

Development overlay — bind API to host loopback

```
services:
  api:
    environment:
      APP_MODE: development
      LOG_LEVEL: debug
    ports:
      - "127.0.0.1:${HOST_HTTP_PORT:-8088}:8080"
```

Listing: compose.dev.yaml (7 lines; copy without page headers or captions).

Production-leaning overlay — stronger API runtime defaults

```
services:
  api:
```

```
  environment:
    APP_MODE: production
    LOG_LEVEL: warning
  ports:
    - "127.0.0.1:${PROD_HTTP_PORT:-8090}:8080"
  read_only: true
  tmpfs:
    - /tmp
  security_opt:
    - no-new-privileges:true
```

Listing: compose.prod.yaml (12 lines; copy without page headers or captions).

In this example, the production override retains base limits, healthcheck, and dependency wiring while changing APP_MODE, LOG_LEVEL, read_only, tmpfs, security_opt, and published host port. A production-leaning overlay is not itself a deployment approval: add HTTPS ingress, backups, monitored SQL transactions, and credential rotation before real use.

Explicitly fictional secret fixture — never commit a real value

```
synthetic-demo-only-not-production
```

Listing: secrets/db_password.txt (1 lines; copy without page headers or captions).

The top-level secrets.db_password.file field points to a file on the Compose host. Only db has a service-level secrets grant, and PostgreSQL reads its path using POSTGRES_PASSWORD_FILE. Linux-container Compose secrets are exposed as mounted files under /run/secrets; in ordinary single-host Compose they are backed by host files, not magically encrypted at rest. Protect host permissions, backups, and secret rotation. _FILE is an image-specific convention: confirm each image supports it.

Repository exclusion rules — demonstration

```
.env
secrets/
__pycache__/
*.pyc
```

Listing: .gitignore (4 lines; copy without page headers or captions).

Validation before deployment

At the keyboard

```
docker compose -f compose.yaml -f compose.dev.yaml config --quiet
docker compose -f compose.yaml -f compose.dev.yaml config --services
docker compose -f compose.yaml -f compose.dev.yaml config --images
docker compose -f compose.yaml -f compose.dev.yaml config --hash
docker compose -f compose.yaml -f compose.prod.yaml config --quiet
```

config --quiet validates the Compose model without starting containers; it does not prove the base image exists, a secret can be read at runtime, a volume is writable, or network and business tests pass. The resolved config can be reviewed as YAML or JSON, but do not export a real production model without redacting sensitive information.

Full local run / safe stop

At the keyboard

```
python verify_reference.py  # optional offline source-fixture check
docker compose -f compose.yaml -f compose.dev.yaml up -d --build --wait
# Open http://127.0.0.1:8088/health in your browser.
docker compose -f compose.yaml -f compose.dev.yaml ps
docker compose -f compose.yaml -f compose.dev.yaml down
```

Windows PowerShell can call python or py -3 for the optional offline check. The Docker Compose commands are the same wherever the Compose V2 plugin is installed. Use docker compose down without -v to retain the named PostgreSQL volume. Always back up real databases independently; a surviving volume is not a verified backup.

Offline fixture contract — verify_reference.py

```
"""Offline source-fixture checks; not a replacement for docker compose config."""
from pathlib import Path
import subprocess
import sys

ROOT = Path(__file__).resolve().parent
NEEDED = [
    "compose.yaml", "compose.dev.yaml", "compose.prod.yaml",
    ".env.example", "api.env", ".gitignore",
    "api/Dockerfile", "api/main.py", "secrets/db_password.txt",
]


def verify():
    errors = [f"missing file: {name}" for name in NEEDED if not (ROOT / name).is_file()]
    if errors:
        return errors
    base = (ROOT / "compose.yaml").read_text(encoding="utf-8")
    dev = (ROOT / "compose.dev.yaml").read_text(encoding="utf-8")
    prod = (ROOT / "compose.prod.yaml").read_text(encoding="utf-8")
    secret = (ROOT / "secrets/db_password.txt").read_text(encoding="utf-8").strip()
    if "/run/secrets/db_password" not in base:
        errors.append("database secret path not declared")
    if "service_healthy" not in base:
        errors.append("database readiness gate not declared")
    if "127.0.0.1:" not in dev or "127.0.0.1:" not in prod:
```

```
        errors.append("host-only dev/prod publishing absent")
    if "read_only: true" not in prod:
        errors.append("production read-only root filesystem absent")
    if secret != "synthetic-demo-only-not-production":
        errors.append("sample secret must be synthetic; do not test with a real secret")
    return errors


if __name__ == "__main__":
    failures = verify()
    print("OFFLINE FIXTURE CHECK:", "FAIL" if failures else "PASS")
    for reason in failures:
        print(" -", reason)
    if failures:
        sys.exit(1)
    print("Next: docker compose -f compose.yaml -f compose.dev.yaml config --quiet")
    # This check never requires a Docker daemon or contacts a registry.
```

Listing: verify_reference.py (43 lines; copy without page headers or captions).

Failure guide: from symptom to evidence

Symptom	Likely check	Safer next action
config reports missing variable	--env-file and ${VAR:?} expansion	Set required non-secret value; rerun config
API starts before DB is usable	db healthcheck and depends_on gate	Check pg_isready and real SQL smoke test
Port is unexpectedly public	Resolved ports under multiple -f files	Remove extra override; bind 127.0.0.1
Secret does not mount	Top-level file exists, per-service grant	Verify path and permissions; do not echo value
Data disappears after teardown	Volume name / down -v usage	Restore from known backup; never claim down -v is safe
Memory/CPU contention	limits, actual stats, workload	Adjust limits after measurements

DeployBoard appendix acceptance receipt

Check	Pass evidence to record
1. Compose parser	Both dev and prod config --quiet complete with exit 0.
2. Deliberate publication	Only one API loopback port; DB has no host port.
3. Startup readiness	db health reaches healthy before API, under tested engine.
4. App observation	/health returns 200; no claim of SQL transaction test.
5. Environment priority	Container APP_LABEL and APP_MODE match resolved model.
6. Secret boundary	Only db receives db_password; no real value in source control.
7. Data durability	db_data survives down and service recreation (not down -v).
8. Operations receipt	Docker / Compose versions, logs, resolved hash, and rollback recorded.

> ***RELEASE GATE*** *BUILD the fixture, VERIFY the resolved model and observed runtime behavior, BREAK only a disposable copy, REPAIR from the known-good Compose files, then SHIP a reviewed configuration with a*

> *written rollback and backup plan. A successful YAML parse alone is not a deployment sign-off.*

Appendix E — Complete DeployBoard Source Code and Reproducibility Kit

DeployBoard is the evolving service from the preceding chapters. This appendix gives you a reproducible single-host implementation with FastAPI, PostgreSQL, Redis, an API worker, Compose, a loopback-only proxy, tests, CI checks, and a restore drill. You will find the exact public source of every file in this appendix; no separate repository is necessary.

OWNER'S CONTRACT The packaged project is a teaching reference, not an assertion of production certification. It can be tested locally without Docker for request logic; real PostgreSQL, Compose, image pulls, networking, TLS, vulnerability and backup checks require you to run them and record actual evidence.

Reference freeze: 20 October 2026 • Python 3.12 • Compose v2.20+ • library versions recorded below. Image tags must be resolved to SHA256 digests on the release host. Avoid placing real credentials in screenshots, printed files, or commits.

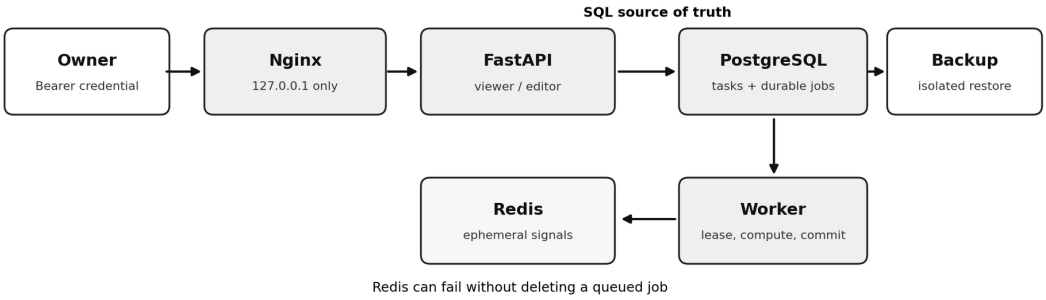


Figure E.1. The API authorizes reads and writes; PostgreSQL owns durable jobs; the worker reports results; Redis carries only optional non-sensitive signals.

E.1 Version-Locked Manifest and Canonical Project File Tree

Version-lock the pieces you control: Python packages use exact == versions; the image tags below specify release lines, and MANIFEST.json records SHA256 checksums of all 40 public files. Versioned container tags can be republished; a final deployment must pin inspected OCI digests and note the platform. MANIFEST.json verifies file integrity within the kit, but is not signed and is not independent proof of origin.

Layer	Reference decision	Freeze and verify
Python	3.12; requirements.txt	Pin dependencies; collect resolved package metadata
Containers	Python 3.12.11, PostgreSQL 17.6, Redis 7.4.6, Nginx 1.28.0	Resolve and record SHA256 per platform
Compose	2.20+ recommended minimum	Validate with installed docker compose version
Database	001_core.sql → 002_indexes.sql	Never rewrite migrations already applied
Files	41 public file digests	Run verify_release.py; protect manifest separately
Credentials	Generated locally, excluded	Restrict host ACL and rotate after exposure

Canonical directory map

DeployBoard_Complete_Kit/
.dockerignore
.env.example
.github/workflows/ci.yml
.gitignore
Dockerfile
MANIFEST.json
README.md
app/__init__.py
app/auth.py
app/cache.py
app/config.py
app/db.py
app/main.py
app/models.py
app/schemas.py
compose.dev.yaml
compose.prod.yaml
compose.yaml
docs/README_FIRST.txt
docs/release_receipt.txt
nginx/default.conf
requirements-dev.txt
requirements.txt
samples/curl_reference.sh
samples/expected_report.txt
samples/tasks.json
scripts/__init__.py
scripts/backup.sh
scripts/deploy_local.sh
scripts/generate_secrets.py
scripts/lock_manifest.py
scripts/migrate.py
scripts/restore_drill.sh
scripts/verify_release.py
sql/001_core.sql
sql/002_indexes.sql
tests/conftest.py
tests/test_api.py
tests/test_security.py
tests/test_structure.py

worker/__init__.py
worker/runner.py

Public version and hash manifest

MANIFEST.json | 51 lines

FILE SHA256 c2252b7df15b… · exact source listing · no line numbers


```
{
 "compose_minimum": "2.20",
 "dependency_lock": "requirements.txt and requirements-dev.txt exact versions",
 "files": {
  ".dockerignore": "1e2ac75b4f6c0fc60c258cf9d87ee814ef535ebdae7159d9eda1a492b3d3de57",
  ".env.example": "051a57b9077abbdb7473b288a21cd8575827c09289da403b302da229827987e3",
  ".github/workflows/ci.yml": "e90e7c6495329d1796b0d05682e42a2889e0714e4180f9b2826dd6e15978af51",
  ".gitignore": "c4d09a07db18e3a03969aacc9d4ed1dd631659c9f846d60b10e955e2f9a888ba",
  "Dockerfile": "a9a0a4d841cf5ee81b781566fee37130c47cdea882734eeffc557ce716c0896b",
  "README.md": "be419ee185bdedb6c25459988789376e60e02777b2817b4707949f1745fe864b",
  "app/__init__.py": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "app/auth.py": "0f21c3590475d0d364854e0831b776ca4f6a3055117bebbc7469467e15aab9e2",
  "app/cache.py": "8cb6a267a595eadb23e3eda155a7be24341182c88945df5128720c29d674f936",
  "app/config.py": "d5d79ee49d603c3cf9ed3ef27361e84673ef08aae9bb58d4bd7f6c5efbd524ef",
  "app/db.py": "3a0f6ff0f376bb3440a92f3054ca07611778e0a95091dfbdf5a792f0d2b949f0",
  "app/main.py": "ebfffbea17b1c5ffc71423558173a64ab2050cf4a5b17661c4a367b50026add6",
  "app/models.py": "d56f3d0331f4044f431478fffff64b9005ef2d06dc2eedfebeaf1b4fca1c4c7c",
  "app/schemas.py": "a7cb67a74eb58e7f2e27e9d364f404f7c1af0eb60786403696041b28e84225a9",
  "compose.dev.yaml": "0cfb5152de9628b9638a9eff4ca0e0503ef7bd476728aa2a87d3d3ba63f51adb",
  "compose.prod.yaml": "18a5977dac7ba2839ed2f666d2e075cdada25be7a36f90a0ee06d147cacd6818",
  "compose.yaml": "0011f5d7671c9216576b2b778b71aefc05cee385b9ff3d6ff9748acc41d7f73a",
  "docs/README_FIRST.txt": "fe19819bc2d3b9514bd118aea338ace6a1eb6cfae4004cc10e2e50932d31bada",
  "docs/release_receipt.txt": "46766d36a6c3a91c757926c526e046bbdc3f98901ef78795de6ddb5b012ab549",
  "nginx/default.conf": "1ec63ecf16cbd4f8133d0148f025e86b9ce784220c726c41a8bd0a7372da4a3d",
  "requirements-dev.txt": "da81696fe67c40c761f2b473d1a208395047aa959bda21917282bcc230747c38",
  "requirements.txt": "2979b8c8f84ec5878556ad58b4060ad72a54f656b3dfc030911d970f37a1c3b4",
  "samples/curl_reference.sh": "53f1380f0c6b79a8d94f9d3996acbe9fff4f535d2a881bf143f8786ac4423958",
  "samples/expected_report.txt": "600cccb519268ba06047924c3a08e8054eb3e9fbed79de891502bf8ef0c35d05",
  "samples/tasks.json": "4d889cae9f200ee6b24630157fa67a0cadebd8b5fe678baa722e8b8bf92aa035",
  "scripts/__init__.py": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "scripts/backup.sh": "67ec0847820397bf4e3056b2f5434e090e1a04e2afae91f9c060069d6f3c60ad",
  "scripts/deploy_local.sh": "6902bde6e75f1ffff3757b0418daedf39e9e6b72199fe6d2ad0fec5694a29baa",
  "scripts/generate_secrets.py": "deb8e8da594129d9859e630d9d45bdd4ad1efe90e4915ad3a7808b9685fa2f02",
  "scripts/lock_manifest.py": "99a7bee5b7b21e3aa041d537b7b25f5bdf05eef3c9bde72e8c0f429673f12396",
  "scripts/migrate.py": "9dd0c35687a18aca247cd35929101016752d410a150e3a1e346a618ca247a422",
  "scripts/restore_drill.sh": "cb818eade21f0bc646fb9322d8efbd7cccd5ea7f55e8ccb11ae71a3d28ad8863",
  "scripts/verify_release.py": "3ba0890d55f6f52b7a8b08a93aa2ee9b13bf472f2c070fa4e6c105ef2685951f",
  "sql/001_core.sql": "7f43b083a061b284454d9e7ea87b8623a5a9954288a59ce154515c78c07e2f29",
  "sql/002_indexes.sql": "84959a5af54c8dde68d0ab03246013a1d1b11775c089ddd7cdd75fcb14a9a8e0",
  "tests/conftest.py": "9bc571ae13009d856cd621924116144b7e64e39ea621ba356f986ac356e55e0b",
  "tests/test_api.py": "62c2d38417f7602484e5f5b3eb35da6538176223611e879f2c82dd73f48a679d",
  "tests/test_security.py": "04304c10b55133c062a3b1a4f34354918dcf5e0c72e40db7c8252b451794b5c9",
  "tests/test_structure.py": "8aba2ff45d7010e8224553f74317cbbb88559cc48505577e7527585d0046472e",
  "worker/__init__.py": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "worker/runner.py": "b42496b14842c7fd0592bbd59bce4cebfd583b208606856016f765c59bfe8a55"
 },
 "image_policy": "versioned tags; resolve SHA256 image digests during live freeze",
 "manifest_version": 1,
 "purpose": "Offline reproducibility; not a signed release",
 "python_reference": "3.12"
}
```


E.2 Complete FastAPI Source: Routes, Models, Authentication, and Configuration

The API uses a narrow bearer-token distinction: viewers can list tasks and inspect jobs; editors can create tasks, change state, and request reports. Tokens are read from Compose-mounted secret files and compared with constant-time equality. This is a deliberate two-role lab, not a replacement for a full identity provider. The database remains authoritative for both tasks and report-job requests.

Endpoint	Scope	Expected behavior
GET /healthz	Public	Liveness only; no database claim
GET /readyz	Public	Returns 503 if database cannot be reached
GET /v1/tasks	Viewer/editor	Lists stored task rows
POST /v1/tasks	Editor	Validates and commits a task
PATCH /v1/tasks/{id}	Editor	Changes only task state
POST /v1/jobs/report	Editor	Requires stable Idempotency-Key
GET /v1/jobs/{id}	Viewer/editor	Shows queued, running, or completed job

app/__init__.py | 0 lines

FILE SHA256 e3b0c44298fc… · exact source listing · no line numbers

(empty Python package marker; create an empty file)

app/config.py | 32 lines

FILE SHA256 d5d79ee49d60… · exact source listing · no line numbers

```
"""Explicit configuration. Real credentials are read from mounted files."""
import os
from functools import lru_cache
from pathlib import Path
from sqlalchemy.engine import URL

def read_secret(variable: str) -> str:
    path = os.getenv(variable, "")
    if not path:
        raise RuntimeError(f"{variable} must point to an existing secret file")
    p = Path(path)
    if not p.is_file():
        raise RuntimeError(f"Missing secret file for {variable}")
    value = p.read_text(encoding="utf-8").strip()
    if len(value) < 32 or value.lower().startswith("replace-"):
        raise RuntimeError(f"{variable} secret is missing or too short")
    return value

def database_url() -> str | URL:
    if os.getenv("DATABASE_URL"):
        # A local SQLite URL is used by the offline test suite only.
        return os.environ["DATABASE_URL"]
    return URL.create(
        "postgresql+psycopg",
        username=os.getenv("DB_USER", "deployboard"),
        password=read_secret("DB_PASSWORD_FILE"),
        host=os.getenv("DB_HOST", "db"),
        port=int(os.getenv("DB_PORT", "5432")),
        database=os.getenv("DB_NAME", "deployboard"),
    )
```

app/db.py | 15 lines

FILE SHA256 3a0f6ff0f376… · exact source listing · no line numbers

```
"""Database engine and request-scoped sessions."""
from functools import lru_cache
from sqlalchemy import create_engine
from sqlalchemy.orm import Session
from app.config import database_url

@lru_cache(maxsize=1)
def engine():
    return create_engine(database_url(), pool_pre_ping=True, future=True)
```

```
    def session_scope():
        with Session(engine()) as session:
            yield session
```

app/models.py | 30 lines

FILE SHA256 d56f3d0331f4… · exact source listing · no line numbers

```
    """Canonical ORM shape; SQL migrations define the production schema."""
    from datetime import date, datetime, timezone
    from sqlalchemy import Date, DateTime, ForeignKey, Index, Integer, String, Text, UniqueConstraint
    from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column

    class Base(DeclarativeBase):
        pass

    class Task(Base):
        __tablename__ = "tasks"
        id: Mapped[int] = mapped_column(Integer, primary_key=True)
        title: Mapped[str] = mapped_column(String(160), nullable=False)
        priority: Mapped[str] = mapped_column(String(10), nullable=False, default="medium")
        state: Mapped[str] = mapped_column(String(12), nullable=False, default="open")
        due_date: Mapped[date | None] = mapped_column(Date, nullable=True)
        created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), default=lambda: datetime.now(timezone.utc), nullable=False)

    class Job(Base):
        __tablename__ = "jobs"
        __table_args__ = (UniqueConstraint("idempotency_key", name="uq_jobs_request"), Index("ix_jobs_state", "state"))
        id: Mapped[int] = mapped_column(Integer, primary_key=True)
        kind: Mapped[str] = mapped_column(String(24), nullable=False)
        state: Mapped[str] = mapped_column(String(16), nullable=False, default="queued")
        idempotency_key: Mapped[str] = mapped_column(String(90), nullable=False)
        result: Mapped[str | None] = mapped_column(Text, nullable=True)
        claimed_at: Mapped[datetime | None] = mapped_column(DateTime(timezone=True), nullable=True)
        created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True), default=lambda: datetime.now(timezone.utc), nullable=False)
```

app/schemas.py | 14 lines

FILE SHA256 a7cb67a74eb5… · exact source listing · no line numbers

```
    """Input validation shared by HTTP endpoints."""
    from datetime import date
    from typing import Literal
    from pydantic import BaseModel, Field

    class TaskIn(BaseModel):
        title: str = Field(min_length=1, max_length=160)
        priority: Literal["low", "medium", "high"] = "medium"
        due_date: date | None = None

    class TaskPatch(BaseModel):
        state: Literal["open", "done"]
```

app/auth.py | 25 lines

FILE SHA256 0f21c3590475… · exact source listing · no line numbers

```
    """Two read-only / read-write bearer scopes; never log tokens."""
    import hmac
    from fastapi import Header, HTTPException
    from app.config import read_secret

    def require_role(authorization: str | None = Header(default=None), write: bool = False) -> str:
        if not authorization or not authorization.startswith("Bearer "):
            raise HTTPException(status_code=401, detail="Bearer token required")
        supplied = authorization.removeprefix("Bearer ").strip()
        read_token = read_secret("APP_TOKEN_READ_FILE")
        write_token = read_secret("APP_TOKEN_WRITE_FILE")
        if hmac.compare_digest(supplied, write_token):
            return "editor"
        if not write and hmac.compare_digest(supplied, read_token):
            return "viewer"
        raise HTTPException(status_code=403, detail="Forbidden")

    def reader(authorization: str | None = Header(default=None)) -> str:
        return require_role(authorization, write=False)

    def editor(authorization: str | None = Header(default=None)) -> str:
        return require_role(authorization, write=True)
```

app/main.py | 96 lines

FILE SHA256 ebfffbea17b1… · exact source listing · no line numbers

```
"""DeployBoard FastAPI: role checks, task CRUD, durable job requests."""
from datetime import datetime, timezone
from fastapi import Depends, FastAPI, Header, HTTPException, status
from sqlalchemy import func, select
from sqlalchemy.exc import IntegrityError, SQLAlchemyError
from sqlalchemy.orm import Session
from app.auth import editor, reader
from app.cache import notify
from app.db import engine, session_scope
from app.models import Job, Task
from app.schemas import TaskIn, TaskPatch

def task_out(task: Task) -> dict:
    return {"id": task.id, "title": task.title, "priority": task.priority,
            "state": task.state, "due_date": str(task.due_date) if task.due_date else None}

def job_out(job: Job) -> dict:
    return {"id": job.id, "state": job.state, "kind": job.kind, "result": job.result}

def create_app() -> FastAPI:
    app = FastAPI(title="DeployBoard", version="1.0.0")

    @app.get("/healthz")
    def healthz():
        return {"status": "alive"}

    @app.get("/readyz")
    def readyz():
        try:
            with engine().connect() as connection:
                connection.execute(select(1))
            return {"status": "ready"}
        except SQLAlchemyError:
            raise HTTPException(status_code=503, detail="database unavailable")

    @app.get("/v1/tasks")
    def list_tasks(_role: str = Depends(reader), db: Session = Depends(session_scope)):
        return [task_out(t) for t in db.scalars(select(Task).order_by(Task.id))]

    @app.post("/v1/tasks", status_code=status.HTTP_201_CREATED)
    def add_task(data: TaskIn, _role: str = Depends(editor), db: Session = Depends(session_scope)):
        task = Task(title=data.title, priority=data.priority, due_date=data.due_date, state="open")
        db.add(task)
        db.commit()
        db.refresh(task)
        notify("tasks", {"id": task.id, "event": "created"})
        return task_out(task)

    @app.patch("/v1/tasks/{task_id}")
    def change_task(task_id: int, data: TaskPatch,
                    _role: str = Depends(editor), db: Session = Depends(session_scope)):
        task = db.get(Task, task_id)
        if task is None:
            raise HTTPException(status_code=404, detail="Task not found")
        task.state = data.state
        db.commit()
        db.refresh(task)
        notify("tasks", {"id": task.id, "event": "changed"})
        return task_out(task)

    @app.post("/v1/jobs/report", status_code=202)
    def request_report(idempotency_key: str | None = Header(default=None, alias="Idempotency-Key"),
                       _role: str = Depends(editor), db: Session = Depends(session_scope)):
        if not idempotency_key or not 8 <= len(idempotency_key) <= 90:
            raise HTTPException(status_code=422, detail="Idempotency-Key must be 8-90 characters")
        existing = db.scalar(select(Job).where(Job.idempotency_key == idempotency_key))
        if existing:
            return job_out(existing)
        job = Job(kind="report", idempotency_key=idempotency_key, state="queued")
        db.add(job)
        try:
            db.commit()
        except IntegrityError:
            db.rollback()
```

```
            existing = db.scalar(select(Job).where(Job.idempotency_key == idempotency_key))
            if existing:
                return job_out(existing)
            raise
        db.refresh(job)
        notify("jobs", {"id": job.id, "event": "queued"})
        return job_out(job)

    @app.get("/v1/jobs/{job_id}")
    def get_job(job_id: int, _role: str = Depends(reader), db: Session = Depends(session_scope)):
        job = db.get(Job, job_id)
        if job is None:
            raise HTTPException(status_code=404, detail="Job not found")
        return job_out(job)

    return app

app = create_app()
```

E.3 PostgreSQL Migrations, Redis Integration, and Worker Source

The two SQL migrations create the durable storage contract and index common lookups. A one-shot migration container must finish successfully before the API and worker start. A successful migration log proves that the SQL applied, not that existing data survived a future release; pair it with a database backup and a repeatable restore check.

RECOVERY BOUNDARY Do not put the job itself only in Redis. Here the job row is committed to PostgreSQL before worker polling; Redis carries non-sensitive pub/sub notifications. Redis outages may hide a signal, but do not delete the pending job. The worker recovers a stale running lease after five minutes; a genuine external side effect would need stronger idempotency than this report-only demonstration.

sql/001_core.sql | 17 lines

FILE SHA256 7f43b083a061… · exact source listing · no line numbers

```
CREATE TABLE IF NOT EXISTS tasks (
    id SERIAL PRIMARY KEY,
    title VARCHAR(160) NOT NULL,
    priority VARCHAR(10) NOT NULL CHECK (priority IN ('low', 'medium', 'high')),
    state VARCHAR(12) NOT NULL DEFAULT 'open' CHECK (state IN ('open', 'done')),
    due_date DATE,
    created_at TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE IF NOT EXISTS jobs (
    id SERIAL PRIMARY KEY,
    kind VARCHAR(24) NOT NULL,
    state VARCHAR(16) NOT NULL DEFAULT 'queued',
    idempotency_key VARCHAR(90) NOT NULL UNIQUE,
    result TEXT,
    claimed_at TIMESTAMPTZ,
    created_at TIMESTAMPTZ NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

sql/002_indexes.sql | 2 lines

FILE SHA256 84959a5af54c… · exact source listing · no line numbers

```
CREATE INDEX IF NOT EXISTS ix_jobs_state ON jobs (state);
CREATE INDEX IF NOT EXISTS ix_tasks_priority ON tasks (priority);
```

scripts/migrate.py | 26 lines

FILE SHA256 9dd0c35687a1… · exact source listing · no line numbers

```
"""One-shot ordered migration runner; run under an exclusive deployment lock."""
from pathlib import Path
from sqlalchemy import text
from app.db import engine
```

```
def main() -> None:
    files = sorted(Path("sql").glob("[0-9][0-9][0-9]_*.sql"))
    if not files:
        raise SystemExit("No migrations found")
    with engine().begin() as conn:
        conn.execute(text("CREATE TABLE IF NOT EXISTS schema_migrations (name TEXT PRIMARY KEY)"))
        done = {row[0] for row in conn.execute(text("SELECT name FROM schema_migrations"))}
        for file in files:
            if file.name in done:
                continue
            body = file.read_text(encoding="utf-8")
            for statement in body.split(";"):
                if statement.strip():
                    conn.execute(text(statement.strip()))
            conn.execute(text("INSERT INTO schema_migrations (name) VALUES (:name)"), {"name": file.name})
            print("APPLIED", file.name)

if __name__ == "__main__":
    main()
```

app/cache.py | 17 lines

FILE SHA256 8cb6a267a595… · exact source listing · no line numbers

```
"""Optional Redis pub/sub is telemetry, never the job source of truth."""
import json
import os

def notify(topic: str, payload: dict) -> bool:
    url = os.getenv("REDIS_URL", "")
    if not url:
        return False
    try:
        import redis  # Imported lazily so SQLite-only unit tests need no Redis.
        client = redis.Redis.from_url(url, socket_connect_timeout=1, socket_timeout=1)
        client.publish(f"deployboard:{topic}", json.dumps(payload))
        return True
    except Exception:
        # Failure to publish an observation must never roll back a DB transaction.
        return False
```

worker/__init__.py | 0 lines

FILE SHA256 e3b0c44298fc… · exact source listing · no line numbers

(empty Python package marker; create an empty file)

worker/runner.py | 49 lines

FILE SHA256 b42496b14842… · exact source listing · no line numbers

```
"""Poll durable PostgreSQL job rows; Redis is optional telemetry only."""
import os
import time
from datetime import datetime, timedelta, timezone
from sqlalchemy import select
from sqlalchemy.orm import Session
from app.cache import notify
from app.db import engine
from app.models import Job, Task

def execute_one() -> int | None:
    now = datetime.now(timezone.utc)
    with Session(engine()) as db:
        stmt = (select(Job)
                .where((Job.state == "queued") |
                       ((Job.state == "running") & (Job.claimed_at < now - timedelta(minutes=5))))
                .order_by(Job.id).limit(1))
        if engine().dialect.name == "postgresql":
            stmt = stmt.with_for_update(skip_locked=True)
        job = db.scalar(stmt)
        if job is None:
            return None
        job.state = "running"
        job.claimed_at = now
        db.commit()
        job_id = job.id
        try:
            # Count the committed task rows in the same durable database.
            from sqlalchemy import func
```

```
            count = db.scalar(select(func.count(Task.id)).where(Task.state == "open"))
            job.result = f"DeployBoard report: {count} open tasks"
            job.state = "done"
            db.commit()
            notify("jobs", {"id": job_id, "event": "done"})
        except Exception:
            db.rollback()
            raise  # Lease timeout allows a later retry; never invent success.
        return job_id

def main() -> None:
    while True:
        execute_one()
        time.sleep(float(os.getenv("WORKER_POLL_SECONDS", "2")))

if __name__ == "__main__":
    main()
```

E.4 Dockerfiles, Compose Files, Proxy Configuration, and Secret Templates

Compose defines PostgreSQL, Redis, a migration job, API, worker, and Nginx. Neither database nor Redis publishes a host port. Nginx binds loopback only. The worker and migration service do not receive application bearer tokens. There is intentionally no checked-in secret value: run generate_secrets.py to create three distinct high-entropy strings on the owner’s computer.

LINUX SECRET PERMISSIONS The generated host secrets/ directory is restricted to its owner. Individual files allow reads by the non-root container UID through Compose file mounts. Review permissions and Docker Desktop ACL behavior on your platform. Do not mistake a file mount or a secret name for encryption at rest.

The production override adds read-only filesystems and no-new-privileges for application services. It is not a publicly exposed, TLS-secured reverse proxy. Before Internet use, add a hardened authentication/TLS edge, policy enforcement, a secret-management backend, image digest pins, monitoring, and a threat review.

Dockerfile | 13 lines

FILE SHA256 a9a0a4d841cf… · exact source listing · no line numbers

```
FROM python:3.12.11-slim-bookworm
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
WORKDIR /srv/deployboard
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt \
    && groupadd --system app && useradd --system --gid app --home-dir /nonexistent app
COPY app ./app
COPY worker ./worker
COPY sql ./sql
COPY scripts ./scripts
USER app
EXPOSE 8080
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

compose.yaml | 97 lines

FILE SHA256 0011f5d7671c… · exact source listing · no line numbers

```
name: deployboard
services:
  db:
    image: ${POSTGRES_IMAGE:-postgres:17.6-alpine}
    environment:
      POSTGRES_USER: deployboard
      POSTGRES_DB: deployboard
```

```
    POSTGRES_PASSWORD_FILE: /run/secrets/db_password
  secrets: [db_password]
  volumes: [db_data:/var/lib/postgresql/data]
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U deployboard -d deployboard"]
    interval: 10s
    timeout: 3s
    retries: 8
    start_period: 20s
  networks: [backend]
  restart: unless-stopped

redis:
  image: ${REDIS_IMAGE:-redis:7.4.6-alpine}
  command: ["redis-server", "--save", "", "--appendonly", "no"]
  networks: [backend]
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 10s
    timeout: 3s
    retries: 5
  restart: unless-stopped

migrate:
  build:
    context: .
    dockerfile: Dockerfile
  command: ["python", "-m", "scripts.migrate"]
  environment: &db_environment
    DB_HOST: db
    DB_PORT: "5432"
    DB_USER: deployboard
    DB_NAME: deployboard
    DB_PASSWORD_FILE: /run/secrets/db_password
    REDIS_URL: redis://redis:6379/0
  secrets: [db_password]
  depends_on:
    db: {condition: service_healthy}
  networks: [backend]
  restart: "no"

api:
  build: {context: ., dockerfile: Dockerfile}
  command: ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
  environment:
      <<: *db_environment
    APP_TOKEN_READ_FILE: /run/secrets/viewer_token
    APP_TOKEN_WRITE_FILE: /run/secrets/editor_token
  secrets: [db_password, viewer_token, editor_token]
  depends_on:
    migrate: {condition: service_completed_successfully}
    redis: {condition: service_healthy}
  networks: [backend]
  expose: ["8080"]
  healthcheck:
    test: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://127.0.0.1:8080/readyz', timeout=2)"]
    interval: 10s
    timeout: 3s
    retries: 5
  restart: unless-stopped

worker:
  build: {context: ., dockerfile: Dockerfile}
  command: ["python", "-m", "worker.runner"]
  environment: *db_environment
  secrets: [db_password]
  depends_on:
    migrate: {condition: service_completed_successfully}
    redis: {condition: service_healthy}
  networks: [backend]
  restart: unless-stopped

proxy:
  image: ${NGINX_IMAGE:-nginx:1.28.0-alpine}
  ports: ["127.0.0.1:${HTTP_PORT:-8088}:80"]
  volumes:
    - ./nginx/default.conf:/etc/nginx/conf.d/default.conf:ro
```

```
    depends_on:
      api: {condition: service_healthy}
    networks: [backend]
    restart: unless-stopped

networks:
  backend: {internal: true}
volumes:
  db_data: {}
secrets:
  db_password: {file: ./secrets/db_password.txt}
  viewer_token: {file: ./secrets/viewer_token.txt}
  editor_token: {file: ./secrets/editor_token.txt}
```

compose.dev.yaml | 6 lines

FILE SHA256 0cfb5152de96… · exact source listing · no line numbers

```
services:
  api:
    environment:
      APP_ENV: development
  proxy:
    ports: ["127.0.0.1:${HTTP_PORT:-8088}:80"]
```

compose.prod.yaml | 16 lines

FILE SHA256 18a5977dac7b… · exact source listing · no line numbers

```
# Single-host hardening overlay. Add an authenticated TLS edge before public exposure.
services:
  api:
    environment:
      APP_ENV: production
    read_only: true
    tmpfs: [/tmp]
    security_opt: ["no-new-privileges:true"]
  worker:
    read_only: true
    tmpfs: [/tmp]
    security_opt: ["no-new-privileges:true"]
  proxy:
    read_only: true
    tmpfs: [/var/cache/nginx, /var/run]
    security_opt: ["no-new-privileges:true"]
```

nginx/default.conf | 14 lines

FILE SHA256 1ec63ecf16cb… · exact source listing · no line numbers

```
server {
    listen 80;
    server_name _;
    server_tokens off;
    client_max_body_size 128k;
    location / {
        proxy_pass http://api:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Request-ID $request_id;
        proxy_connect_timeout 3s;
        proxy_read_timeout 30s;
    }
}
```

.env.example | 5 lines

FILE SHA256 051a57b9077a… · exact source listing · no line numbers

```
# Copy to .env for a private development workstation.
POSTGRES_IMAGE=postgres:17.6-alpine
REDIS_IMAGE=redis:7.4.6-alpine
NGINX_IMAGE=nginx:1.28.0-alpine
HTTP_PORT=8088
```

.dockerignore | 13 lines

FILE SHA256 1e2ac75b4f6c… · exact source listing · no line numbers

```
.git
.github
.venv
.pytest_cache
__pycache__
*.pyc
```

```
secrets
backups
.env
.env.*
tests
samples
*.zip
```

.gitignore | 8 lines

FILE SHA256 c4d09a07db18… · exact source listing · no line numbers

```
.env
.venv/
__pycache__/
.pytest_cache/
secrets/*.txt
backups/
*.sqlite3
*.pyc
```

scripts/generate_secrets.py | 22 lines

FILE SHA256 deb8e8da5941… · exact source listing · no line numbers

```
"""Create brand-new local test secrets; never print values or overwrite."""
import secrets
from pathlib import Path

def main() -> None:
    target = Path("secrets")
    target.mkdir(exist_ok=True)
    target.chmod(0o700)  # Host directory blocks other local users.
    for name in ("db_password.txt", "viewer_token.txt", "editor_token.txt"):
        dest = target / name
        if dest.exists():
            print("KEPT", name)
            continue
        with dest.open("x", encoding="utf-8") as out:
            out.write(secrets.token_urlsafe(36) + "\n")
        dest.chmod(0o644)  # Readable by non-root containers; parent stays 0700.
        print("CREATED", name, "(value hidden)")

if __name__ == "__main__":
    main()
```

E.5 Automated Tests, CI/CD Workflows, and Deployment/Recovery Scripts

The test fixture creates temporary secret files and a disposable SQLite database; HTTP tests run through FastAPI's TestClient. This isolates application behavior from the Docker Engine and from PostgreSQL-specific migration claims. CI repeats the offline tests and validates the Compose model after generating test-only secrets. A real release requires separate, recorded live infrastructure checks.

BACKUPS ARE NOT RESTORES The backup script captures a PostgreSQL custom-format dump privately. The restore drill checks that an isolated deployboard_restorecheck database does not already exist, then restores there. It deliberately does not overwrite the live deployboard database or automatically delete the verification environment.

tests/conftest.py | 33 lines

FILE SHA256 9bc571ae1300… · exact source listing · no line numbers

```
"""Real FastAPI HTTP integration against temporary SQLite database."""
import pytest
from fastapi.testclient import TestClient
from app.db import engine
from app.models import Base
from app.main import create_app
```

```
@pytest.fixture
def env(tmp_path, monkeypatch):
    viewer = tmp_path / "viewer.txt"
    editor = tmp_path / "editor.txt"
    viewer.write_text("v" * 48, encoding="utf-8")
    editor.write_text("e" * 48, encoding="utf-8")
    monkeypatch.setenv("APP_TOKEN_READ_FILE", str(viewer))
    monkeypatch.setenv("APP_TOKEN_WRITE_FILE", str(editor))
    monkeypatch.setenv("DATABASE_URL", f"sqlite:///{tmp_path / 'test.sqlite3'}")
    monkeypatch.delenv("REDIS_URL", raising=False)
    engine.cache_clear()
    Base.metadata.create_all(engine())
    yield TestClient(create_app())
    engine().dispose()
    engine.cache_clear()

@pytest.fixture
def viewer():
    return {"Authorization": "Bearer " + "v" * 48}

@pytest.fixture
def editor():
    return {"Authorization": "Bearer " + "e" * 48}
```

tests/test_api.py | 81 lines

FILE SHA256 62c2d38417f7… · exact source listing · no line numbers

```
"""Audit-facing HTTP behavioral tests."""
from worker.runner import execute_one

def test_liveness_and_readiness(env):
    assert env.get("/healthz").json()["status"] == "alive"
    assert env.get("/readyz").json()["status"] == "ready"

def test_unauthenticated_list_denied(env):
    assert env.get("/v1/tasks").status_code == 401

def test_malformed_token_denied(env):
    assert env.get("/v1/tasks", headers={"Authorization": "Bearer nope"}).status_code == 403

def test_viewer_can_read(env, viewer):
    assert env.get("/v1/tasks", headers=viewer).json() == []

def test_viewer_cannot_write(env, viewer):
    assert env.post("/v1/tasks", headers=viewer, json={"title": "No"}).status_code == 403

def test_editor_creates_and_reads_task(env, editor, viewer):
    created = env.post("/v1/tasks", headers=editor,
                       json={"title": "Approve plan", "priority": "high"})
    assert created.status_code == 201
    assert created.json()["id"] == 1
    assert env.get("/v1/tasks", headers=viewer).json()[0]["priority"] == "high"

def test_bad_priority_rejected(env, editor):
    assert env.post("/v1/tasks", headers=editor,
                    json={"title": "Bad", "priority": "emergency"}).status_code == 422

def test_missing_title_rejected(env, editor):
    assert env.post("/v1/tasks", headers=editor, json={"title": ""}).status_code == 422

def test_task_change_requires_editor(env, editor, viewer):
    env.post("/v1/tasks", headers=editor, json={"title": "Review"})
    assert env.patch("/v1/tasks/1", headers=viewer, json={"state": "done"}).status_code == 403
    assert env.patch("/v1/tasks/1", headers=editor, json={"state": "done"}).json()["state"] == "done"

def test_missing_task_returns_404(env, editor):
    assert env.patch("/v1/tasks/404", headers=editor, json={"state": "done"}).status_code == 404


def test_idempotency_key_required(env, editor):
    assert env.post("/v1/jobs/report", headers=editor).status_code == 422
```

```
def test_viewer_cannot_queue_job(env, viewer):
    assert env.post("/v1/jobs/report", headers={**viewer, "Idempotency-Key": "stable-00001"}).status_code == 403

def test_duplicate_job_returns_same_id(env, editor):
    h = {**editor, "Idempotency-Key": "stable-00001"}
    a = env.post("/v1/jobs/report", headers=h).json()
    b = env.post("/v1/jobs/report", headers=h).json()
    assert a["id"] == b["id"]
    assert a["state"] == "queued"

def test_worker_completes_one_durable_job(env, editor, viewer):
    env.post("/v1/tasks", headers=editor, json={"title": "A"})
    env.post("/v1/tasks", headers=editor, json={"title": "B"})
    queued = env.post("/v1/jobs/report", headers={**editor, "Idempotency-Key": "report-20261020"}).json()
    assert execute_one() == queued["id"]
    done = env.get(f"/v1/jobs/{queued['id']}", headers=viewer).json()
    assert done["state"] == "done"
    assert done["result"] == "DeployBoard report: 2 open tasks"
    assert execute_one() is None

def test_job_lookup_missing(env, viewer):
    assert env.get("/v1/jobs/999", headers=viewer).status_code == 404
```

tests/test_structure.py | 49 lines

FILE SHA256 8aba2ff45d70… · exact source listing · no line numbers

```
"""Offline configuration and file-contract tests."""
import json
import subprocess
import sys
from pathlib import Path
import yaml

ROOT = Path(__file__).resolve().parents[1]

def test_compose_services():
    spec = yaml.safe_load((ROOT / "compose.yaml").read_text())
    assert set(spec["services"]) == {"db", "redis", "migrate", "api", "worker", "proxy"}

def test_proxy_local_bind_only():
    spec = yaml.safe_load((ROOT / "compose.yaml").read_text())
    assert all(str(p).startswith("127.0.0.1:") for p in spec["services"]["proxy"]["ports"])

def test_db_not_public():
    spec = yaml.safe_load((ROOT / "compose.yaml").read_text())
    assert "ports" not in spec["services"]["db"]
    assert "ports" not in spec["services"]["redis"]

def test_api_waits_for_migration():
    spec = yaml.safe_load((ROOT / "compose.yaml").read_text())
    assert spec["services"]["api"]["depends_on"]["migrate"]["condition"] == "service_completed_successfully"

def test_secrets_not_in_images():
    build = (ROOT / "Dockerfile").read_text()
    assert "COPY secrets" not in build and "ARG PASSWORD" not in build

def test_migration_files_are_ordered():
    assert sorted(x.name for x in (ROOT / "sql").glob("[0-9]*.sql")) == ["001_core.sql", "002_indexes.sql"]

def test_expected_output_fixture():
    assert "3 open tasks" in (ROOT / "samples/expected_report.txt").read_text()
    assert len(json.loads((ROOT / "samples/tasks.json").read_text())) == 3

def test_manifest_checks():
    result = subprocess.run([sys.executable, str(ROOT / "scripts/verify_release.py")],
                            cwd=ROOT, capture_output=True, text=True)
    assert result.returncode == 0, result.stdout + result.stderr
```

tests/test_security.py | 24 lines

FILE SHA256 04304c10b551… · exact source listing · no line numbers

```
"""Non-secret source and upgrade hygiene checks."""
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
```

```
def test_no_real_secret_files_shipped():
    assert not list((ROOT / "secrets").glob("*.txt"))

def test_manifest_excludes_secrets():
    assert "secrets/" not in (ROOT / "MANIFEST.json").read_text()

def test_requirements_exact_pins():
    for line in (ROOT / "requirements.txt").read_text().splitlines():
        if line and not line.startswith("#"):
            assert "==" in line

def test_archive_restore_targets_separate_db():
    script = (ROOT / "scripts/restore_drill.sh").read_text()
    assert "deployboard_restorecheck" in script
    assert "-d deployboard_restorecheck" in script
```

.github/workflows/ci.yml | 17 lines

FILE SHA256 e90e7c649532… · exact source listing · no line numbers

```
name: DeployBoard checks
on:
  push:
  pull_request:
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - run: python -m pip install -r requirements-dev.txt
      - run: python -m pytest -q
      - run: python scripts/verify_release.py
      - run: python scripts/generate_secrets.py
      - run: docker compose config --quiet
```

scripts/deploy_local.sh | 14 lines

FILE SHA256 6902bde6e75f… · exact source listing · no line numbers

```
#!/bin/sh
set -eu
# Deliberately local single-host deployment. Do not expose this stack publicly.
test -f .env || { echo 'STOP: copy .env.example to .env first' >&2; exit 1; }
for name in db_password viewer_token editor_token; do
  test -s "secrets/${name}.txt" || {
    echo "STOP: missing secrets/${name}.txt (run generate_secrets.py)" >&2
    exit 1
  }
done
docker compose config --quiet
docker compose up -d --build
docker compose ps
echo 'Inspect readiness, API permissions, job outcomes, and release receipt before approval.'
```

scripts/backup.sh | 12 lines

FILE SHA256 67ec08478203… · exact source listing · no line numbers

```
#!/bin/sh
set -eu
mkdir -p backups
umask 077
stamp=$(date -u +%Y%m%dT%H%M%SZ)
file="backups/deployboard-${stamp}.dump"
# PostgreSQL custom-format archive, created inside DB container and streamed to host.
docker compose exec -T db sh -c \
  'export PGPASSWORD="$(cat /run/secrets/db_password)"; exec pg_dump -h 127.0.0.1 -U deployboard -d deployboard -Fc' > "$file"
test -s "$file"
printf 'Backup created: %s\n' "$file"
printf '%s\n' 'Keep this private, encrypted at rest, and test restoration separately.'
```

scripts/restore_drill.sh | 19 lines

FILE SHA256 cb818eade21f… · exact source listing · no line numbers

```
#!/bin/sh
set -eu
archive=${1:?Usage: scripts/restore_drill.sh backups/file.dump}
test -s "$archive"
```

```
# NEVER restore into the live database. Use a disposable, distinct database.
present=$(docker compose exec -T db sh -c \
  'export PGPASSWORD="$(cat /run/secrets/db_password)"; psql -h 127.0.0.1 -U deployboard -d postgres -tAc "SELECT count(*) FROM
pg_database WHERE datname = '\''deployboard_restorecheck'\''"')
if [ "$present" != "0" ]; then
  echo 'STOP: restorecheck database exists or cannot be safely ruled out' >&2
  exit 1
fi
# Require operator verification that no previous drill DB exists.
docker compose exec -T db sh -c \
  'export PGPASSWORD="$(cat /run/secrets/db_password)"; createdb -h 127.0.0.1 -U deployboard deployboard_restorecheck'
docker compose exec -T db sh -c \
  'export PGPASSWORD="$(cat /run/secrets/db_password)"; pg_restore -h 127.0.0.1 -U deployboard -d deployboard_restorecheck --no-owner
--no-privileges' < "$archive"
docker compose exec -T db sh -c \
  'export PGPASSWORD="$(cat /run/secrets/db_password)"; psql -h 127.0.0.1 -U deployboard -d deployboard_restorecheck -tAc "SELECT
count(*) FROM tasks"'
echo 'Inspect the isolated restorecheck database. Remove it deliberately after verification.'
```

E.6 Pinned Dependency Files, Sample Records, and Test Fixtures

Python package names and direct versions are explicit rather than left floating. They are reference pins; refresh through a controlled dependency audit and record the refreshed lock and test receipt. Samples contain only fictional DeployBoard tasks. The shell fixture is intended for Unix/macOS/WSL2; on native PowerShell use the equivalent HTTP requests with Invoke-RestMethod.

requirements.txt | 5 lines

FILE SHA256 2979b8c8f84e… · exact source listing · no line numbers

```
fastapi==0.128.2
uvicorn==0.48.0
SQLAlchemy==2.0.50
psycopg[binary]==3.2.10
redis==6.4.0
```

requirements-dev.txt | 4 lines

FILE SHA256 da81696fe67c… · exact source listing · no line numbers

```
-r requirements.txt
pytest==9.0.2
httpx==0.28.1
PyYAML==6.0.3
```

samples/tasks.json | 5 lines

FILE SHA256 4d889cae9f20… · exact source listing · no line numbers

```
[
  {"title":"Review DeployBoard deployment plan","priority":"high","due_date":"2026-10-22"},
  {"title":"Inspect PostgreSQL backup receipt","priority":"medium","due_date":null},
  {"title":"Write release notes","priority":"low","due_date":null}
]
```

samples/expected_report.txt | 1 lines

FILE SHA256 600cccb51926… · exact source listing · no line numbers

```
DeployBoard report: 3 open tasks
```

samples/curl_reference.sh | 13 lines

FILE SHA256 53f1380f0c6b… · exact source listing · no line numbers

```
#!/bin/sh
set -eu
# First run: python scripts/generate_secrets.py
EDITOR=$(cat secrets/editor_token.txt)
VIEWER=$(cat secrets/viewer_token.txt)
BASE=http://127.0.0.1:8088
curl -fsS "$BASE/healthz"
curl -fsS -H "Authorization: Bearer $VIEWER" "$BASE/v1/tasks"
curl -fsS -X POST "$BASE/v1/tasks" \
  -H "Authorization: Bearer $EDITOR" -H 'Content-Type: application/json' \
```

```
  -d '{"title":"Review DeployBoard deployment plan","priority":"high"}'
curl -fsS -X POST "$BASE/v1/jobs/report" \
  -H "Authorization: Bearer $EDITOR" -H 'Idempotency-Key: release-report-001'
```

E.7 Expected Outputs, Commands, and Release Verification Evidence

Do not confuse the sample output below with a transcript of real container execution. The offline acceptance fixture expects a report with two open tasks in one targeted test; the standalone three-task sample file expects three. The acceptance proof is the observed outcome from the specific fixture and environment, not a printed success claim.

Proof	Expected observation	What it does not establish
python -m pytest -q	27 passed on the bundled offline suite	PostgreSQL/Redis containers were healthy
python scripts/verify_release.py	PASS: 41 recorded files match	Files were independently signed
docker compose config --quiet	Exit code 0 and no error output	Containers started successfully
GET /healthz	{"status":"alive"}	Database is reachable
GET /readyz	{"status":"ready"} or 503	All external services are safe
Repeat one Idempotency-Key	Identical job id on both requests	External exactly-once delivery
Isolated restore drill	Restored tasks can be counted	Live production recovery complete

For final publication, retain a filled owner receipt with actual Docker Engine and Compose versions, resolved image digests, migration evidence, health-check evidence, failure-injection notes, database restore details, and an explicit local-pilot approval or rejection.

scripts/__init__.py | 0 lines

FILE SHA256 e3b0c44298fc… · exact source listing · no line numbers

(empty Python package marker; create an empty file)

scripts/lock_manifest.py | 32 lines

FILE SHA256 99a7bee5b7b2… · exact source listing · no line numbers

```
"""Write a reproducibility hash inventory of public kit files only."""
import hashlib
import json
from pathlib import Path

def main():
    root = Path(__file__).resolve().parents[1]
    included = {}
    for p in sorted(root.rglob("*")):
        if not p.is_file():
            continue
        rel = p.relative_to(root).as_posix()
        if rel in {"MANIFEST.json", ".env"} or rel.startswith(("secrets/", "backups/")):
            continue
        if any(part.startswith(".") and part not in {".github", ".env.example", ".dockerignore", ".gitignore"}
               for part in p.relative_to(root).parts):
            continue
        if "__pycache__" in rel or rel.endswith((".pyc", ".sqlite3")):
            continue
        included[rel] = hashlib.sha256(p.read_bytes()).hexdigest()
    data = {"manifest_version": 1, "purpose": "Offline reproducibility; not a signed release",
            "python_reference": "3.12", "compose_minimum": "2.20",
            "dependency_lock": "requirements.txt and requirements-dev.txt exact versions",
            "image_policy": "versioned tags; resolve SHA256 image digests during live freeze",
            "files": included}
```

```
    (root / "MANIFEST.json").write_text(json.dumps(data, indent=2, sort_keys=True) + "\n")
    print("WROTE", len(included), "public checksums")

if __name__ == "__main__":
    main()
```

scripts/verify_release.py | 27 lines

FILE SHA256 3ba0890d55f6… · exact source listing · no line numbers

```
"""Offline structural release checks; NOT a substitute for Docker runtime tests."""
import hashlib
import json
import sys
from pathlib import Path

def main() -> None:
    root = Path(__file__).resolve().parents[1]
    manifest = json.loads((root / "MANIFEST.json").read_text(encoding="utf-8"))
    bad = []
    for rel, expected in manifest["files"].items():
        target = root / rel
        if (not target.is_file() or
                hashlib.sha256(target.read_bytes()).hexdigest() != expected):
            bad.append(rel)
    for secret in ("db_password.txt", "viewer_token.txt", "editor_token.txt"):
        if f"secrets/{secret}" in manifest["files"]:
            bad.append("secret in tracked manifest")
    if bad:
        print("FAIL: " + ", ".join(bad))
        sys.exit(1)
    print(f"PASS: {len(manifest['files'])} recorded files match; secret values excluded")

if __name__ == "__main__":
    main()
```

docs/release_receipt.txt | 20 lines

FILE SHA256 46766d36a6c3… · exact source listing · no line numbers

```
DEPLOYBOARD RELEASE RECEIPT (fill with observed evidence; no invented passes)
Date/time (UTC):
Git commit / local file SHA manifest:
Python version:
Docker Engine and Compose versions:
Image names and resolved sha256 digests:
pytest: PASS / FAIL / NOT RUN
release hashes: PASS / FAIL / NOT RUN
compose config: PASS / FAIL / NOT RUN
migration logs: PASS / FAIL / NOT RUN
healthz/readyz: PASS / FAIL / NOT RUN
Viewer write blocked: PASS / FAIL / NOT RUN
Editor create: PASS / FAIL / NOT RUN
Idempotency retry: PASS / FAIL / NOT RUN
Worker reported final status: PASS / FAIL / NOT RUN
DB backup: PASS / FAIL / NOT RUN
Restore to isolated database: PASS / FAIL / NOT RUN
Secret leak audit: PASS / FAIL / NOT RUN
Approved for local pilot? YES / NO
Release owner and decision:
```

E.8 Clean-Workspace Rebuild Without a Download or External Repository

Recreate the canonical tree printed in E.1 in a clean private directory. Copy each source listing into its exact relative path, preserving indentation and final newlines. Create the three empty __init__.py package markers; never copy files from a real production state directory. After saving the public files, run lock_manifest.py to regenerate the local checksum inventory; compare with the book’s example only when source bytes and line endings match.

BUILD — prepare a clean workspace

Make a new DeployBoard_Complete_Kit folder. Use Python 3.12. On Linux/macOS/WSL2 run the following lines in that folder. On Windows PowerShell use Copy-Item .env.example .env, py -3 -m pip, and py -3 scripts/generate_secrets.py. Do not run the backup/restore shell scripts from ordinary PowerShell unless a compatible POSIX shell is available.

```
cp .env.example .env
python scripts/generate_secrets.py
python -m pip install -r requirements-dev.txt
python scripts/lock_manifest.py
python scripts/verify_release.py
python -m pytest -q
```

VERIFY — reject partial or contradictory results

Expected after an unchanged rebuild: 41 public hashes match and 27 tests pass. If they do not, inspect the first named file or failing test rather than changing the expected output. A SHA mismatch may be a whitespace or line-ending difference; a failed permission test is a release blocker.

BREAK → REPAIR — prove controls before deployment

In a disposable copy only, change the editor token to the viewer value and rerun tests: the distinct-role design should fail the viewer-write guard. Restore distinct credentials. In a separate copy, change a task fixture or Compose port from 127.0.0.1 to 0.0.0.0; the checksum/structural suite should reject the alteration. Do not perform mutation tests on the release candidate.

SHIP — start a bounded local pilot and collect real evidence

```
docker compose version
docker compose config --quiet
docker compose up --build -d
docker compose ps
docker compose logs --tail=60 api worker migrate
curl http://127.0.0.1:8088/healthz
```

Run samples/curl_reference.sh for controlled task and job tests; inspect the response before changing state. Execute scripts/backup.sh and scripts/restore_drill.sh against an isolated practice stack; use docs/release_receipt.txt to record results. Only the owner can promote this local pilot after evidence of safety, repeatable deployment, and recovery.

README.md | 27 lines

FILE SHA256 be419ee185bd… · exact source listing · no line numbers

```
# DeployBoard 1.0 — Self-contained reference source kit

This is a **single-host teaching implementation** that follows the *Docker Projects Handbook* DeployBoard case study. Appendix D's minimal standard-library API was a Compose illustration; Appendix E supersedes it with FastAPI, a PostgreSQL schema, and an optional Redis signal channel.

## Minimum kit setup

1. Work on a **new practice folder**, not an existing production project. Keep real credentials out of the book examples.
2. Copy `.env.example` to `.env`.
3. Run `python scripts/generate_secrets.py`; **do not commit** `.env`, files in `secrets/`, or backups. The host `secrets/` directory is mode 0700; mounted secret files are mode 0644 so the non-root container user can read them. On Windows, verify equivalent ACL protection.
4. Run `python -m pip install -r requirements-dev.txt`, then `python -m pytest -q` and `python scripts/verify_release.py`. The offline test suite uses SQLite and is not a PostgreSQL-in-Docker integration test.
5. With Docker Engine and Compose installed: `docker compose config --quiet`, then `sh scripts/deploy_local.sh` (or run `docker compose up -d --build` manually).
6. Check `docker compose ps`, `docker compose logs --tail=60 api worker migrate`, and `curl http://127.0.0.1:8088/healthz`.
7. Run `sh samples/curl_reference.sh` on Unix/WSL2; alternatively perform the commands individually. Confirm no duplicate report jobs from the same idempotency key.
```

8. Create a private backup: `sh scripts/backup.sh`; rehearse restore **into the separate restorecheck database only**: `sh scripts/restore_drill.sh backups/<name>.dump`.

What is and is not proved

- Python tests establish request handling, roles, duplicate protection, worker transitions, and fixture validity with an ephemeral **SQLite** database.
- Compose YAML parsing establishes syntax **structure**, not that images exist or containers are healthy.
- The `migrate` service uses plain SQL migration files; update them incrementally and never edit migrations already applied to real environments.
- Redis publishes **ephemeral telemetry**. PostgreSQL jobs remain the durable queue. Avoid sensitive task content in Redis messages.
- Proxy maps only `127.0.0.1:8088`, **not a public HTTPS endpoint**. To deploy publicly, provide an externally managed TLS and authentication edge, rotation, monitoring, immutable image digests, vulnerability scanning, a restricted host firewall, and protected backups.
- Image version tags and Python package `==` pins are recorded. Tags are not immutable: capture inspected SHA256 digests after pulling approved images and freeze your actual deploy by digest.

Release threshold

Keep `docs/release_receipt.txt` as an operator-completed evidence sheet. A local test passing does **not** certify internet production readiness. Run real PostgreSQL, Docker Compose, network isolation, health, controlled failure, and backup-restore verification before accepting a release.

docs/README_FIRST.txt | 16 lines

FILE SHA256 fe19819bc2d3… · exact source listing · no line numbers

```
DEPLOYBOARD COMPLETE KIT - START HERE
1. Copy .env.example to .env (never commit .env).
2. python scripts/generate_secrets.py (creates secret values; do not use book examples as credentials).
3. python -m pip install -r requirements-dev.txt (local offline code tests; install resolves packages over network).
4. python -m pytest -q (SQLite/unit integration, no Docker needed).
5. python scripts/verify_release.py (hashes the public project files).
6. docker compose config --quiet (requires Docker/Compose available locally).
7. sh scripts/deploy_local.sh (requires Docker Engine; images may be pulled).
8. docker compose ps; curl http://127.0.0.1:8088/healthz
9. Run scripts/curl_reference.sh, inspect results, record your receipt.
10. scripts/backup.sh, scripts/restore_drill.sh backups/... for owner-controlled PostgreSQL drill.

SECURITY: ports bind loopback only; NOT an Internet deployment. Add authenticated HTTPS edge,
network policies, monitoring, backups, non-root controls, and image digests before public rollout.
Use only synthetic data. Never commit real secrets or database backups.
Runtime proof must be collected on your machine; the build environment cannot launch Docker Engine.
```

Appendix F — Cross-Platform Terminal and File-System Guide

Docker commands may be portable, but the shell that launches them and the filesystem that supplies their files are not. This appendix helps you translate a working DeployBoard command into Windows PowerShell, a WSL 2 Linux shell, macOS Terminal, or a Linux host without changing the application's Linux-container contract.

Working rule Prefer relative Compose paths, immutable application configuration, LF text files used by Linux, and non-destructive checks. Use native host-specific commands only at the shell boundary.

> **SCOPE** Examples use Linux containers on Docker Desktop or Docker Engine. They do not describe Windows-container Dockerfiles or Windows-container permissions. Commands assume the current directory contains the supplied mini-lab files.

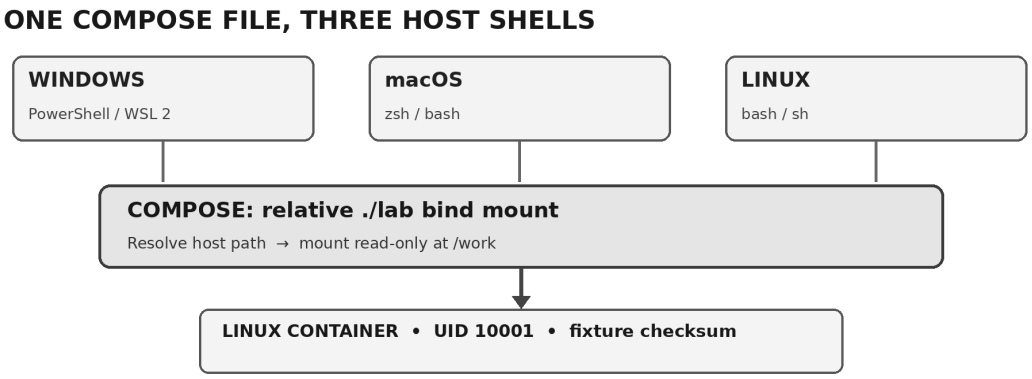


Figure F.1 One Compose specification, three host environments, one Linux-container contract.

F.1 Bash, PowerShell, and macOS Command Differences

Bash on Linux, Bash inside WSL, and the default zsh on recent macOS installations share most of the shell syntax used below. Windows PowerShell is different: quoting, environment assignment, pipeline behavior, commands such as curl, and paths need attention. Choose one shell per code block; do not mix PowerShell lines into a Bash script.

Goal	Bash / zsh (Linux, macOS, WSL)	PowerShell (Windows)
Find current directory	pwd	Get-Location
List files, including hidden	ls -la	Get-ChildItem -Force
Enter project folder	cd ./deployboard	Set-Location .\deployboard
Create a directory	mkdir -p ./lab/output	New-Item -ItemType Directory -Force ./lab/output
Show file contents	cat ./compose.yaml	Get-Content .\compose.yaml
Check file existence	test -f ./compose.yaml	Test-Path .\compose.yaml -PathType Leaf
Set session environment	export APP_MODE=demo	$env:APP_MODE = "demo"
Run Python program	python3 verify_portable.py	py -3 verify_portable.py (or python verify_portable.py)
Display exit status	echo $?	$LASTEXITCODE for native apps
Find lines in output	grep -n "read_only" compose.yaml	Select-String -Path compose.yaml -Pattern read_only

PowerShell commands such as Get-Content and New-Item are cmdlets; Bash utilities such as cat and mkdir are executable programs or shell utilities. In PowerShell, an alias might shadow curl on older installations. For HTTP tests use curl.exe on Windows, or use Invoke-WebRequest and inspect its status explicitly. Use docker compose config --quiet for a syntax-only check: dumping the resolved configuration in shared logs could reveal injected values.

Terminal-neutral Docker commands (each shell, from the lab root)

```
docker version
docker compose version
docker compose config --quiet
docker compose build inspector
docker compose run --rm --no-deps inspector
docker compose down
```

CROSS-SHELL GOTCHA A shell variable assigned with $env:APP_MODE in PowerShell affects that PowerShell session and subprocesses. APP_MODE=demo docker compose ... is valid in Bash/zsh but is not the equivalent PowerShell syntax. Docker commands themselves are generally unchanged.

Set a single-run value in Bash or zsh

```
APP_MODE=staging docker compose run --rm --no-deps inspector
```

Set a value in PowerShell, then run the same Docker command

```
$env:APP_MODE = 'staging'
docker compose run --rm --no-deps inspector
Remove-Item Env:APP_MODE -ErrorAction SilentlyContinue
```

If your shell cannot find docker, check whether Docker Desktop is running, whether the Docker CLI is installed on your PATH, and whether WSL integration is enabled for the distribution you use. Do not reinstall or delete containers before examining docker context ls and docker info.

F.2 Path Mapping, Bind Mounts, and Filesystem Performance

A bind mount attaches an existing file or directory from the Docker daemon host to a path inside a container. The container path is Linux-style even when the person launching Compose uses PowerShell on Windows. Bind mounts are right for changing source and fixtures during development; named volumes are generally a better fit for PostgreSQL files and other application-managed persistent data.

Host/workspace	Typical source path	Container target
Windows PowerShell	C:\Users\you\deployboard\lab	/work
WSL 2 Ubuntu	/home/you/deployboard/lab	/work
macOS Terminal	/Users/you/deployboard/lab	/work
Linux Bash	/home/you/deployboard/lab	/work

Rather than hardcoding any of these absolute paths, Compose should resolve ./lab relative to its project file. Run commands from the lab root. When specifying multiple -f Compose files, remember that merged relative paths are anchored to the first/base Compose file, not necessarily the directory of an override.

FILE compose.yaml

```
name: appendix-f-portability
services:
```

```
inspector:
  build:
    context: .
    dockerfile: Dockerfile
  image: appendix-f-inspector:local
  user: "10001:10001"
  read_only: true
  environment:
    APP_MODE: ${APP_MODE:-demo}
  volumes:
    - type: bind
      source: ./lab
      target: /work
      read_only: true
      bind:
        create_host_path: false
  working_dir: /work
  command: ["python", "/work/inspect_environment.py"]
```

Why this pattern is safer Long-syntax bind mounting uses read_only: true and create_host_path: false. If the host source is missing, Compose reports a problem instead of quietly creating an empty directory that masks your container image files. No database or Redis ports are published.

Situation	Symptom	First useful check
Missing host source	Bind mount cannot be created	Confirm ./lab exists next to compose.yaml.
Wrong project directory	Unexpected empty or wrong files	Run pwd / Get-Location and compose config --quiet.
Remote Docker context	Bind mount points to wrong machine	Check docker context show; bind sources live on daemon host.
Slow code bind mount	High file-scan latency	On WSL 2, move source to Linux filesystem; for DB use named volume.
Case mismatch	Works on one host, fails on another	Match file name case exactly; avoid case-only differences.

For Windows with WSL 2, store frequently accessed Linux-container source under /home/... inside the WSL distribution, not /mnt/c/...; Docker documents both better bind-mount performance and more reliable file-change events there. On macOS, Docker Desktop shares host paths into its Linux VM; VirtioFS and related file-sharing options can affect performance. Benchmark the actual workload rather than assuming a single filesystem path or setting wins everywhere.

> **PROTECT DATABASE DATA** Do not convert DeployBoard PostgreSQL named volumes into cross-platform source-code bind mounts merely for convenience. Check backups and access rules before moving persistent storage. Never use docker compose down --volumes for this appendix lab or the production application.

Inspect bind mount metadata without changing data

```
docker compose config --quiet
docker compose run --rm --no-deps inspector
```

```
docker compose down
```

The lab's fixture is read-only. A successful run prints PASS: fixture readable, mode, UID/GID, working directory, and a SHA-256 checksum. It does not expose a secret or write files to the host.

F.3 User IDs, Ownership, Line Endings, and Executable Files

For Linux containers, file access is governed by the effective numeric UID/GID, mode bits, mount settings, and host filesystem behavior. An application may run as UID 10001 and read a 0644 fixture successfully while failing to write to a host bind-mounted directory owned by another user. Container USER and Compose user do not automatically reassign ownership of a bind mount.

FILE Dockerfile

```
# syntax=docker/dockerfile:1
FROM python:3.12-alpine
RUN addgroup -S -g 10001 app && \
   adduser -S -D -H -u 10001 -G app app
WORKDIR /work
COPY --chown=10001:10001 lab/ /work/
USER 10001:10001
CMD ["python", "/work/inspect_environment.py"]
```

The Dockerfile creates an unprivileged app identity and uses COPY --chown to assign ownership to image-layer files. Compose runs the inspector with the same numeric UID. The live read-only bind mount covers /work, so the actual fixture's host-side permissions still matter. With Docker Desktop, the host-to-VM sharing layer can change how ownership and mode bits appear; verify access from inside the container instead of assuming an exact host-to-container mapping.

Check / repair	Linux or macOS shell	PowerShell / WSL note
Show Linux UID/GID	id -u; id -g	In WSL shell use id; Windows ACLs differ.
Show POSIX file mode	ls -l ./lab/fixture.txt	Use WSL ls -l for Linux metadata.
Set ordinary file mode	chmod 644 ./lab/fixture.txt	Use a WSL/Linux shell when POSIX modes matter.
Set script executable	chmod +x ./scripts/check.sh	Use WSL/Linux; PowerShell .ps1 is not a POSIX executable.
Inspect Windows ACL	Not applicable	Get-Acl .\lab\fixture.txt

AVOID INDISCRIMINATE FIXES Do not repair a permission error with chmod -R 777 or by running the entire stack as root. Identify the mount, effective UID, target file or directory, and whether write access is actually required. Change only that path after confirming you are not modifying a production database volume.

Text files are another portability boundary. Unix shebang scripts and many Linux tools expect LF line endings. A CRLF shell script may fail with errors resembling "/bin/sh^M: bad interpreter." The simplest prevention is a project .gitattributes policy, not manual edits to every developer's Git installation.

FILE .gitattributes

```
* text=auto
*.sh text eol=lf
*.py text eol=lf
*.yaml text eol=lf
*.ps1 text eol=crlf
Dockerfile text eol=lf
```

Check and repair one shell script (Bash, Linux/macOS/WSL)

```
file scripts/check.sh
# If needed, remove carriage returns without editing its logic:
python3 -c "from pathlib import Path; p=Path('scripts/check.sh');
p.write_bytes(p.read_bytes().replace(bytes([13,10]), bytes([10])))"
chmod +x scripts/check.sh
```

Do not run the newline-conversion command on binary files. A shebang script also requires the executable bit when launched as ./scripts/check.sh; invoking sh scripts/check.sh avoids that permission requirement, but not the LF requirement. On Windows, Git may not reliably preserve POSIX execute bits from the working tree; use git update-index --chmod=+x scripts/check.sh when working in a Git repository and verify the committed mode before shipping.

FILE scripts/check.sh

```
#!/bin/sh
set -eu
printf 'Host shell: %s\n' "${SHELL:-sh}"
printf 'Project directory: %s\n' "$(pwd)"
printf 'Docker CLI: '
docker --version
printf 'Compose plugin: '
docker compose version
```

FILE scripts/check.ps1

```
$ErrorActionPreference = 'Stop'
Write-Output "Project directory: $(Get-Location)"
docker --version
if ($LASTEXITCODE -ne 0) { throw 'Docker CLI failed' }
docker compose version
if ($LASTEXITCODE -ne 0) { throw 'Compose plugin failed' }
```

Expected check result Both scripts print the project directory and Docker/Compose CLI versions when those tools are installed. A missing CLI or stopped Docker daemon is a genuine failure, not something the scripts should silently ignore.

F.4 Portable Environment Variables and File Permission Checks

Use the host shell to set nonsensitive inputs, and let Compose interpolate them into the configuration. The top-level .env file supports Compose interpolation, while service env_file supplies variables to the container; the concepts are not interchangeable. Shell-exported variables may override .env interpolation values. For credentials, use file-mounted Compose secrets rather than environment strings that may surface in process details or configuration output.

Task	Bash / zsh	PowerShell
Set value this session	export APP_MODE=demo	$env:APP_MODE = "demo"
Print value	printf "%s\n" "$APP_MODE"	$env:APP_MODE
Unset value	unset APP_MODE	Remove-Item Env:APP_MODE -ErrorAction SilentlyContinue
Check file exists	test -f ./lab/fixture.txt	Test-Path .\lab\fixture.txt -PathType Leaf
Check nonsecret file SHA-256	sha256sum ./lab/fixture.txt (Linux) shasum -a 256 ./lab/fixture.txt (macOS)	Get-FileHash .\lab\fixture.txt -Algorithm SHA256
Read POSIX owner inside container	docker compose run --rm inspector	docker compose run --rm inspector

PowerShell’s $LASTEXITCODE tracks native-process exit status. Bash uses $? immediately after the command. For reliable scripts, always check exit status at the point of failure rather than assuming a message containing “PASS” is sufficient. When showing configurations to colleagues, do not paste docker compose config output if secrets or tokens might be interpolated into it.

FILE lab/inspect_environment.py

```
#!/usr/bin/env python3
"""Read-only portability probe; deliberately prints no secret values."""
import hashlib
import os
from pathlib import Path

fixture = Path("/work/fixture.txt")
if not fixture.is_file():
    raise SystemExit("FAIL: read-only fixture is missing")
raw = fixture.read_bytes()
if b"\r\n" in raw:
    raise SystemExit("FAIL: fixture has CRLF line endings")
if raw != b"DeployBoard portability fixture\n":
    raise SystemExit("FAIL: unexpected fixture contents")
print("PASS: fixture readable")
print("mode:", os.environ.get("APP_MODE", "(unset)"))
print("uid:", os.getuid(), "gid:", os.getgid())
print("cwd:", os.getcwd())
print("sha256:", hashlib.sha256(raw).hexdigest())
```

FILE lab/fixture.txt

```
DeployBoard portability fixture
```

FILE verify_portable.py

```
"""Offline, non-destructive verification of the appendix lab files."""
from pathlib import Path
import sys

base = Path(__file__).resolve().parent
need = ["compose.yaml", "Dockerfile", ".gitattributes",
        "lab/inspect_environment.py", "lab/fixture.txt",
        "scripts/check.sh", "scripts/check.ps1"]
for relative in need:
    if not (base / relative).is_file():
        raise SystemExit("FAIL: missing " + relative)
fixture = (base / "lab/fixture.txt").read_bytes()
if fixture != b"DeployBoard portability fixture\n":
    raise SystemExit("FAIL: fixture changed or has CRLF")
for relative in ("scripts/check.sh", "lab/inspect_environment.py",
                 "compose.yaml", "Dockerfile"):
    if b"\r" in (base / relative).read_bytes():
        raise SystemExit("FAIL: unexpected CR in " + relative)
compose = (base / "compose.yaml").read_text(encoding="utf-8")
assert "create_host_path: false" in compose
assert "read_only: true" in compose
assert 'user: "10001:10001"' in compose
assert "./lab" in compose
print("PASS: file tree, LF rules and safe Compose patterns")
```

Offline verification, no Docker required

```
python verify_portable.py
```

Expected: PASS: file tree, LF rules and safe Compose patterns. The verifier refuses missing files, modified fixture contents, and CR bytes in Linux-facing files. It is intentionally not a substitute for Docker Compose validation or a runtime permission test.

Full cross-platform acceptance run, from the lab root

```
docker compose config --quiet
docker compose build inspector
docker compose run --rm --no-deps inspector
docker compose down
```

Acceptance proof: the container exits with code zero; reports UID and GID 10001, current directory /work, and PASS: fixture readable; the checksum matches the host fixture; and no persistent data or public ports were created. If a check fails, record your host OS, shell, docker version, docker context show, the exact command, and the nonsecret error text. That incident record makes the failure reproducible without disclosing credentials.

PRODUCTION BOUNDARY The lab uses a convenience Python base-image tag so the exercise can be followed independently. The actual DeployBoard release workflow in Chapters 25–32 uses recorded image digests, credential-safe configuration, separate production volumes, and release approval gates.

Portable lab file tree compose.yaml, Dockerfile, .gitattributes, verify_portable.py, README.txt, lab/inspect_environment.py, lab/fixture.txt, scripts/check.sh, and scripts/check.ps1. The ZIP includes all nine files, and every file's contents are printed above or specified directly in this appendix.

APPENDIX G —Troubleshooting Decision Trees

DeployBoard has six roles to distinguish: db (PostgreSQL), redis (notifications), migrate (one-shot schema setup), api (FastAPI), worker (durable jobs), and proxy (Nginx). A container can be running while the application is unavailable; a migration can exit successfully while all long-running services remain healthy. This reference gives you an ordered way to decide which fault domain to investigate.

USE THIS FIRST *Work in the DeployBoard directory from Appendix E. Record the Compose project name, active files, release ID and first failing timestamp. Do not paste real secret values or complete unredacted logs into public reports. No step here authorizes deleting named volumes or replacing a live database.*

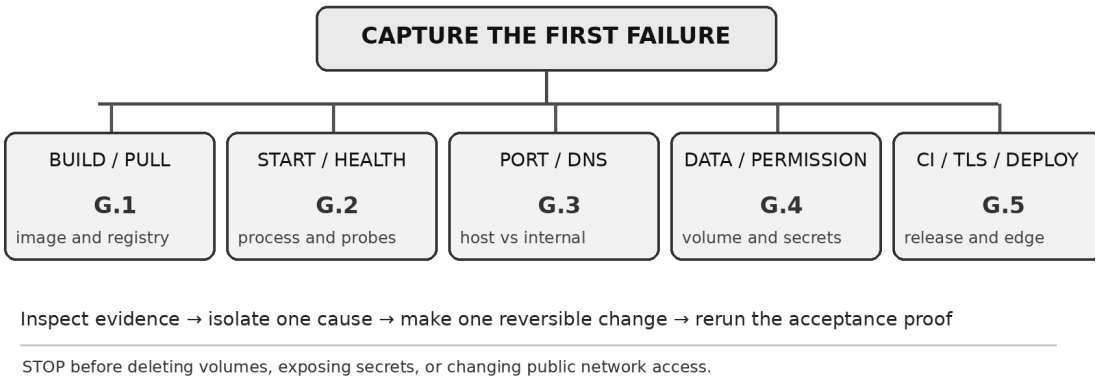


Figure G.1 Evidence-led routing: five distinct failure domains, one controlled recovery method.

The four-command read-only intake

RUN IN THE ACTUAL DEPLOYBOARD PROJECT DIRECTORY

```
docker compose version
docker compose config --quiet
docker compose ps --all
docker compose logs --no-color --tail 60
```

Read the first failure, not the last cascade. config --quiet validates the resolved model without dumping interpolated values. ps --all includes exited one-shot containers. Logs can contain private data: view locally, redact before sharing, and preserve the first decisive error.

G.1 Build Fails or Image Cannot Be Pulled

Start by separating a Compose-model error, a local Docker build failure, and a registry pull problem. Each has a different fix; none benefits from deleting images or disabling authentication indiscriminately.

Question / observed evidence	Next safe action	Accept only when…
config --quiet fails before a build?	Confirm active -f files, project directory, interpolation, YAML keys.	The exact intended file set validates.
Dockerfile says COPY failed or file not found?	Check Dockerfile , .dockerignore , build context and file casing.	The source exists inside the build context.
Registry returns 401/403 or denied?	Check image spelling, account entitlement and registry token scope; use docker login privately if authorized.	The exact image tag pulls with least-privilege access.
Manifest unknown / no matching platform?	Inspect chosen tag and architecture; use a supported image for host platform.	Tag exists and supports target architecture.
DNS, TLS or timeout during pull?	Check host connectivity, proxy and CA trust; do not bypass certificate checks.	Retried pull succeeds without insecure flags.

ISOLATE BUILD FROM PULL

```
docker compose config --quiet
docker compose build --progress plain api
docker compose pull db redis proxy
docker image ls
```

For this project, api , worker , and migrate share a local Dockerfile ; db , redis , and proxy use registry images. Do not assume a registry login error is a Dockerfile error. The default image tags are examples; pin approved digests before a production release.

BREAK → REPAIR *Reproduce a harmless missing-file failure only in a disposable copy: rename one COPY source, run the build, restore its filename, and rebuild. A missing file should be fixed in source, not with --no-cache as the first response.*

G.2 Container Exits, Restarts, or Becomes Unhealthy

Classify the state before touching the service. The migrate container is expected to finish with exit code 0; api , db , redis , worker , and proxy are intended to remain running. A restart policy does not itself repair a failing healthcheck or broken dependency.

State or signal	Interpretation / next branch	Recovery proof
migrate exited (0)	Normal only if the schema and dependent service checks pass.	api reaches readyz ; migration logs show success.
migrate exited nonzero	Schema, credentials, or database readiness: inspect migration and database logs before retry.	Single successful migration, no unexpected schema divergence.
api running but unhealthy	Distinguish failed /readyz , wrong probe, or slow startup; verify database connectivity.	Healthy state plus authenticated API read.
Exit 137 / OOMKilled=true	Check host memory, container constraints and traffic; do not assume code crash.	No further OOM under the same bounded load.

State or signal	Interpretation / next branch	Recovery proof
Repeated restart with exit 1	Inspect first exception and required secret mounts; fix startup cause.	Stable running state and correct logs.
worker healthy? no probe defined	Worker has no Compose healthcheck in the reference; running state is not proof of job delivery.	A queued job changes state exactly once and has evidence.

READ-ONLY STATE AND PROBE INVESTIGATION

```
docker compose ps --all
docker compose logs --no-color --tail 100 api migrate db
docker compose ps -q api
# Replace <container-id> with the preceding returned ID:
docker inspect <container-id> --format '{{json .State}}'
docker compose exec -T db pg_isready -U deployboard -d deployboard
```

The `docker inspect` format is shell-friendly on Bash; on Windows PowerShell, paste the same command carefully and substitute the actual container ID. Never expect a containerized healthcheck to restart a sick process automatically. First solve the failing probe or dependency, then perform one controlled recreate if configuration changed.

CONCLUSIVE DISTINCTION *`/healthz` proves that the API process responds; `/readyz` also queries the database and can return HTTP 503. A successful `/healthz` alone is not a production readiness proof.*

G.3 Port, DNS, and Service Connectivity Errors

Use the correct vantage point. On the single-host DeployBoard reference stack, the host reaches Nginx at `127.0.0.1:8088` by default, while containers on the shared `backend` network reach `api:8080`, `db:5432`, and `redis:6379`. `localhost` inside a container means that container—not the host or its peers.

First observation	Likely fault domain	Check next
Host `127.0.0.1:8088` refuses	Proxy not running, host port already owned, or incorrect publish mapping.	Check `ps --all`, active `.env` `HTTP_PORT` and proxy logs.
Proxy replies 502 / 504	Nginx is reachable; API listener, readiness, timeout or upstream network is wrong.	Check `api` state, logs, and `/readyz` from service.
API says `db` cannot resolve	Wrong service name or not on the same Compose network.	Inspect resolved Compose network memberships.
DNS resolves but connection refused	Target process or port not listening yet; this is not a DNS failure.	Check service readiness and listening address.
`curl` works locally but not remotely	Reference proxy binds to loopback by design.	Do not expose it publicly without a reviewed TLS and auth edge.

HOST TEST / EXPECTED ROUTES

```
# Bash / macOS / Linux:
curl -i http://127.0.0.1:8088/healthz
curl -i http://127.0.0.1:8088/readyz
# PowerShell: use curl.exe to avoid alias differences.
curl.exe -i http://127.0.0.1:8088/readyz
```

CONTAINER NETWORK AND SERVICE-NAME CHECKS

```
docker compose config --services
docker compose ps --all
docker compose exec -T api python -c "import socket; print(socket.getaddrinfo('db',5432))"
docker compose logs --no-color --tail 60 proxy api
```

Expected: `/healthz` returns JSON `{"status":"alive"}` when the web process runs; `/readyz` returns `{"status":"ready"}` only when the database responds. A 401/403 on protected `/v1/tasks` is an authorization finding, not evidence of broken container networking. Do not replace service-name DNS with hard-coded container IPs.

G.4 Volumes, Database Initialization, and Permission Failures

A PostgreSQL volume contains persistent state. An initialization variable is not a repeatable migration mechanism; once an existing volume is populated, changing `POSTGRES_USER` or `POSTGRES_DB` in Compose does not safely rebuild it. In DeployBoard, schema changes belong to the `migrate` job, while the durable PostgreSQL data stays in `db_data`.

Symptom	Safe diagnostic	Do not do this
Database logs show existing data or init skipped	Check whether `db_data` is reused and what schema already exists.	Do not delete the volume to force initialization.
Permission denied on `/var/lib/postgresql/data`	Inspect mount type, filesystem permissions and image UID requirements.	Do not run `chmod -R 777` or change UID blindly.
Missing `/run/secrets/db_password`	Confirm generated secret file exists at host path and Compose secret mapping is active.	Do not print or paste secret contents into logs.
Migration fails after DB healthy	Inspect the first SQL error and previously applied migration version.	Do not replay migrations without examining their idempotency.
Backup file exists but restore fails	Validate archive type and run isolated restore drill on a distinct database.	Never test restore into live DeployBoard data.

READ-ONLY DATABASE AND MOUNT INVESTIGATION

```
docker compose logs --no-color --tail 90 db migrate
docker compose ps --all
docker volume ls
docker compose config --quiet
# Print mount metadata, not volume or secret file contents:
docker inspect <db-container-id> --format '{{json .Mounts}}'
```

The reference kit provides `scripts/backup.sh` and `scripts/restore_drill.sh`. Inspect them before execution. The backup streams a PostgreSQL custom-format dump to a private local file. The drill restores into the separately named `deployboard_restorecheck` database and refuses to reuse a preexisting drill database. The drill is not permission to destroy or overwrite `deployboard`.

RECOVERY GATE *Before making a persistent-data change, confirm the backup archive, its protected storage, the correct Compose project/volume, and a successful isolated restore. Treat* `docker compose down -v` *,* `docker volume prune` *, and unrestricted permission changes as destructive operations—not routine troubleshooting.*

G.5 CI/CD, TLS, and Deployment Troubleshooting

Differentiate build pipeline failure, artifact provenance failure, deployment failure, and public TLS failure. The Appendix E reference GitHub Actions workflow runs Python tests, a release-manifest check, synthetic-secret generation, and `docker compose config --quiet` . It does not, by itself, prove a published image was deployed or that the host has a valid TLS certificate.

Branch / evidence	First responsible check	Release gate
CI fails at Python tests	Reproduce with the pinned requirements; inspect the first failing test.	Same test suite passes without changing expected behavior.
CI fails at `verify_release.py`	Compare manifest hashes to intended changes; investigate drift before relocking.	Provenance changes are reviewed and documented.
Image push returns denied	Check registry destination, login permissions and tag immutability.	Published digest verified before deployment.
TLS name or chain error	Inspect certificate SAN, expiration, trust chain and actual TLS terminator.	HTTPS validates without disabling verification.
Deploy succeeds but site unavailable	Check `ps` , `api` readiness, `proxy` edge and authenticated endpoint.	End-to-end smoke tests and durable jobs pass.
New release regresses	Use recorded prior digest and controlled rollback; preserve database compatibility.	Prior health and acceptance proofs restored.

LOCAL PRE-RELEASE CHECKS (READER-RUN)

```
python -m pytest -q
python scripts/verify_release.py
docker compose config --quiet
docker compose ps --all
curl -i http://127.0.0.1:8088/readyz
```

For the single-host reference, port `8088` is plain HTTP bound to loopback; it is not a public TLS endpoint. To expose DeployBoard externally, first design and validate an authenticated TLS termination edge. Never use `curl -k` or turn off certificate checks to make a bad certificate appear good. Production deployment should use approved image digests, captured release metadata, and backward-compatible database migration planning.

SHIP OR HOLD *A green CI badge and running containers are necessary but insufficient. Ship only when health, database readiness, authentication boundaries, expected records, worker job completion, and rollback evidence all pass against the intended release.*

Reusable Offline Fault-Triage Helper

A small offline triage helper prevents “it seems like DNS” from becoming an unexamined conclusion. Feed it a synthetic or redacted observation with `category` ,

signal , and service ; it returns a single next check and labels it HYPOTHESIS_ONLY . It never calls Docker or reads a credential. The examples below are complete and reproduced in the accompanying ZIP.

CREATE triage.py — COMPLETE PYTHON 3.10+ LISTING

```
#!/usr/bin/env python3
"""DeployBoard offline triage helper. Accepts only sanitized operator observations.
This tool never connects to Docker, a network, or a credential store.
"""
import argparse
import json
from pathlib import Path

CATEGORIES = {'build', 'lifecycle', 'network', 'storage', 'delivery'}
REQUIRED = {'category', 'signal', 'service'}
RULES = {
    ('build', 'denied'): ('CHECK_REGISTRY_AUTH', 'Check image name, registry authorization and token scope.'),
    ('build', 'context'): ('CHECK_BUILD_CONTEXT', 'Check Dockerfile path, COPY sources and .dockerignore.'),
    ('build', 'platform'): ('CHECK_IMAGE_PLATFORM', 'Inspect image architectures and the host platform.'),
    ('lifecycle', 'oom'): ('CHECK_MEMORY', 'Inspect OOMKilled, memory pressure and actual limits.'),
    ('lifecycle', 'unhealthy'): ('CHECK_HEALTH', 'Inspect the failing probe, timeout and dependencies.'),
    ('lifecycle', 'exit'): ('CHECK_EXIT', 'Inspect exit code and first fatal log message.'),
    ('network', 'host-port'): ('CHECK_HOST_PORT', 'Check localhost binding, port owner and published port.'),
    ('network', 'dns'): ('CHECK_SERVICE_DNS', 'Check Compose service name and shared networks.'),
    ('network', 'refused'): ('CHECK_LISTENER', 'Check process listener and readiness before DNS changes.'),
    ('storage', 'permission'): ('CHECK_MOUNT_PERMISSIONS', 'Check mount mode, UID/GID and secret access.'),
    ('storage', 'init'): ('CHECK_DB_STATE', 'Inspect database logs, persisted volume and migrations.'),
    ('delivery', 'ci'): ('CHECK_CI_STAGE', 'Find first failing CI stage and repeat locally.'),
    ('delivery', 'tls'): ('CHECK_TLS_EDGE', 'Check certificate name, trust chain and proxy termination.'),
    ('delivery', 'deploy'): ('CHECK_RELEASE_HEALTH', 'Check release ID, health and reversible rollback.'),
}

def classify(record):
    if not isinstance(record, dict) or not REQUIRED <= record.keys():
        raise ValueError('Missing category, signal, or service')
    category, signal, service = (record[key] for key in ('category', 'signal', 'service'))
    if category not in CATEGORIES or not all(isinstance(v, str) and v for v in (signal, service)):
        raise ValueError('Invalid category, signal, or service')
    if len(signal) > 32 or len(service) > 32:
        raise ValueError('Field is too long')
    code, next_check = RULES.get((category, signal),
        ('INSPECT_MANUALLY', 'No rule matched: preserve evidence and diagnose manually.'))
    return {'status': 'HYPOTHESIS_ONLY', 'service': service,
            'diagnostic_code': code, 'next_check': next_check,
            'warning': 'No live Docker check was performed.'}

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('observation', help='Path to synthetic/sanitized JSON')
    args = parser.parse_args()
    record = json.loads(Path(args.observation).read_text(encoding='utf-8'))
    print(json.dumps(classify(record), indent=2))
```

```
if __name__ == '__main__':
    main()
```

CREATE test_triage.py — COMPLETE REGRESSION SUITE

```
"""Offline regression checks; no Docker installation needed."""
import unittest
from triage import classify

class DiagnosticTests(unittest.TestCase):
    def test_registry_denial(self):
        self.assertEqual(classify({'category':'build','signal':'denied','service':'api'})['diagnostic_code'],
'CHECK_REGISTRY_AUTH')
    def test_context_failure(self):
        self.assertEqual(classify({'category':'build','signal':'context','service':'api'})['diagnostic_code'],
'CHECK_BUILD_CONTEXT')
    def test_unhealthy(self):
        self.assertEqual(classify({'category':'lifecycle','signal':'unhealthy','service':'db'})['diagnostic_code'],
'CHECK_HEALTH')
    def test_oom(self):
        self.assertEqual(classify({'category':'lifecycle','signal':'oom','service':'api'})['diagnostic_code'],
'CHECK_MEMORY')
    def test_dns(self):
        self.assertEqual(classify({'category':'network','signal':'dns','service':'api'})['diagnostic_code'],
'CHECK_SERVICE_DNS')
    def test_host_port(self):
        self.assertEqual(classify({'category':'network','signal':'host-port','service':'proxy'})['diagnostic_code'],
'CHECK_HOST_PORT')
    def test_permissions(self):
        self.assertEqual(classify({'category':'storage','signal':'permission','service':'db'})['diagnostic_code'],
'CHECK_MOUNT_PERMISSIONS')
    def test_database_init(self):
        self.assertEqual(classify({'category':'storage','signal':'init','service':'migrate'})['diagnostic_code'],
'CHECK_DB_STATE')
    def test_tls(self):
        self.assertEqual(classify({'category':'delivery','signal':'tls','service':'proxy'})['diagnostic_code'],
'CHECK_TLS_EDGE')
    def test_ci(self):
        self.assertEqual(classify({'category':'delivery','signal':'ci','service':'api'})['diagnostic_code'],
'CHECK_CI_STAGE')
    def test_unknown_safe(self):
        result=classify({'category':'network','signal':'mystery','service':'api'})
        self.assertEqual(result['diagnostic_code'], 'INSPECT_MANUALLY')
        self.assertEqual(result['status'], 'HYPOTHESIS_ONLY')
    def test_missing(self):
        with self.assertRaises(ValueError): classify({'category':'build','service':'api'})
    def test_invalid(self):
        with self.assertRaises(ValueError): classify({'category':'other','signal':'dns','service':'api'})
    def test_non_text(self):
        with self.assertRaises(ValueError): classify({'category':'build','signal':['denied'],'service':'api'})

if __name__ == '__main__':
    unittest.main()
```

CREATE example_observation.json

```
{
```

```
  "category": "network",
  "signal": "dns",
  "service": "api"
}
```

RUN OFFLINE (WINDOWS: python OR py -3)

```
python -m unittest -q test_triage
python triage.py example_observation.json
```

Expected: 14 tests pass, followed by a JSON reply containing `CHECK_SERVICE_DNS` and `HYPOTHESIS_ONLY`. This is a decision prompt, not a finding that the real Docker service has failed. After diagnosing the actual symptom, record the evidence and verify that the failure no longer occurs.

Final Operator's Decision Receipt

Every repair should leave enough evidence for the next operator to distinguish a fix from a temporary improvement. Copy the following receipt to your private project notes; redact request bodies and credentials. Do not attach database dumps or environment secrets to a support ticket.

COPY-READY INCIDENT RECORD

```
DEPLOYBOARD / APPENDIX G INCIDENT RECEIPT
Timestamp (timezone):
Deployment/release ID (no credentials):
Host OS and Docker/Compose versions:
Affected service and symptom:
Observation source (ps / logs / inspect / curl / CI):
First decisive evidence (redacted):
Most likely fault domain:
Change authorized:
Backup / rollback point confirmed:
Action taken:
Post-change /healthz and /readyz results:
Database migration / job checks:
Regression suite result:
Incident closed by / at:
```

Gate	Proof that closes the incident
Compose model	Intended base/override files pass `config --quiet` with known project identity.
Service state	Expected long-running services stable; `migrate` exits 0 only after successful schema work.
External path	Proxy responds on the authorized host address; `/readyz` returns ready.
Application contract	Reader and editor permissions enforced; expected task/job behavior verified.
Data safety	Backup exists; isolated restore drill proven; no unnecessary volume changes.
Release quality	Offline tests, CI, health probes and rollback evidence reflect the same release.

APPENDIX G OUTCOME *You can locate a failure without guessing, choose the smallest justified repair, and prove that DeployBoard recovered. Stop and request a second review whenever the recovery requires destroying state, exposing private ports, rotating real credentials, or changing live TLS settings.*

Pocket Index: First Command by Symptom

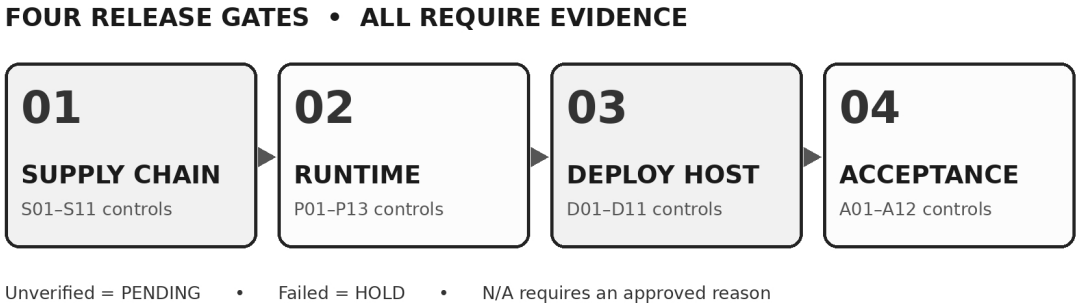
Use this index when a failure arrives without a clear category. Choose one observation and move to the matching section; do not run a collection of fixes at random.

What you see	First discriminating command	Go to
no matching manifest	docker compose pull db (or named image)	G.1: registry and platform
api exited or unhealthy	docker compose ps --all then logs api	G.2: process and health
502 Bad Gateway	docker compose logs --tail 60 proxy api	G.3: proxy to upstream
db cannot resolve	docker compose config --services	G.3: service DNS and network
permission denied at DB path	docker compose logs --tail 80 db	G.4: volume ownership
CI is green but release is down	docker compose ps --all and /readyz	G.5: post-deployment acceptance

STOP CONDITIONS *No verified backup for state changes; unknown Compose project; inconsistent migration evidence; uncertain public exposure; or unavailable previous release. Preserve evidence and escalate before making irreversible changes.*

Appendix H — Security and Production Readiness Checklists

A field-ready sign-off system for the DeployBoard image, containers, production host, and authenticated API. Review the evidence; do not merely check boxes.



How to use this appendix. Treat each line as an acceptance test. Record status PASS, FAIL, PENDING, or N/A; attach a dated and non-secret artifact reference. N/A needs a named approver and reason. A critical failure or missing mandatory proof means HOLD—not a production-ready release.

Status	Meaning	Required action
PASS	Verified against real target	Attach proof and reviewer/date.
FAIL	Acceptance criterion not met	Stop release; create remedial work.
PENDING	Not yet tested or evidence absent	Do not assume success; test first.
N/A	Not applicable to this exact mode	Attach written justification and approval.

Deployment identity to record before beginning: registry digest _____ source revision _____ environment (local/public) _____ UTC review date

_____ evidence folder _____.

H.1 Image Source and Supply Chain Review

Release question: can you explain exactly what code and dependencies are inside the chosen image, who built it, and whether the identified risk is acceptable?

What counts as supply-chain evidence. A successful build is not provenance. An SBOM inventories packages; provenance records build inputs; a signature ties an artifact to a trust identity. None alone proves a release is safe. Record the digest first and make every review artifact refer to that digest.

Review procedure. Use Chapter 20's scanner to generate a dated report, Chapter 25's digest-verification record, and Chapter 26's CI run. The reviewer should check exploitable vulnerabilities, supported base versions, and exception expiry before marking this gate PASS.

ID	Acceptance condition	Required proof / reviewer field
S01	Choose approved base images and record each exact digest.	Signed-off base-image inventory; vendor support and update policy.
S02	Pin the DeployBoard release to a registry@sha256 digest, not a floating tag.	Release receipt and manifest reference match deployment configuration.
S03	Resolve installed package versions and review known vulnerabilities.	Dated scan plus triage; exploitable critical/high findings block absent waiver.
S04	Review every vulnerability exception with owner and expiry.	CVE, exposure analysis, compensating control, approver, expiry.
S05	Produce a readable SBOM for the exact approved release.	SPDX/CycloneDX file linked to digest; retained in evidence store.
S06	Capture BuildKit provenance and check expected builder/source.	Attestation inspected; compare source revision and builder identity.
S07	Verify signing policy where signing is required.	Trusted identity/key and verification record; not merely a signature present.
S08	Keep build secrets out of layers, image history, and build logs.	Review Dockerfile; masked build log; controlled negative scan.
S09	Protect CI from untrusted changes gaining publication credentials.	PR gate has read-only token; publish requires trusted version-tag push.
S10	Require tested, approved source and a passing release quality gate.	CI run URL/ID, commit SHA, tests, review decision.
S11	Capture precise platform coverage, not just a tag label.	Image index/manifest evidence for target linux/amd64 or arm64.

Evidence log for this gate: control ID → PASS / FAIL / PENDING / N/A → non-secret artifact path or report ID → UTC timestamp → reviewer. Do not write passwords or tokens on this sheet.

Record the digest and inspect registry image metadata

```
docker buildx imagetools inspect REGISTRY/IMAGE@sha256:DIGEST
```

```
# Replace placeholders with your actual immutable reference
```

Collect dated scan and attestation evidence

```
trivy image --format json --output scan.json REGISTRY/IMAGE@sha256:DIGEST
# Run your chosen, documented scanner and retain its version
# BuildKit: docker buildx build --sbom=true --provenance=mode=min --push ...
```

Risk decision: scan findings need explicit owner, exploitability assessment, mitigation and expiration. Never equate “no critical CVE” with secure-by-default or treat scan output as a complete runtime audit.

H.2 Least Privilege, Filesystems, Networking, and Secrets

Release question: can compromised application code escalate privileges, write outside intended locations, reach the host daemon, or read unrelated credentials?

Do not mistake declarations for enforced security. A YAML setting is intention; inspect the running container and try the expected allowed workflow. Check the effective application UID and write paths. Official database and proxy images may require different permissions—document specific exceptions, do not force an incompatible universal override.

Secrets boundary. Compose file-mounted secrets are not Swarm-managed encrypted secrets. They still originate from host files. Never print the source or content of a secret in an audit report; record permissions and who can read or rotate it instead.

ID	Acceptance condition	Required proof / reviewer field
P01	Run the API, publisher, and worker with an unprivileged identity.	Inspect effective UID, not only Dockerfile USER text.
P02	Make their root filesystems read-only; allow only necessary writes.	Runtime inspect plus tested /tmp writable tmpfs boundary.
P03	Drop all app capabilities and forbid privilege escalation.	cap_drop ALL and no-new-privileges verified in configuration/runtime.
P04	Avoid privileged mode, host PID/network, devices, and daemon socket mounts.	Runtime mount/device inspection; documented exception required.
P05	Retain only justified proxy binding capability.	NET_BIND_SERVICE proxy exception recorded and functionally verified.
P06	Constrain memory, CPU, and process counts by measurement.	Inspect pids_limit/mem_limit/cpus; load test and OOM result.
P07	Keep DB and Redis on private network with no published ports.	Config audit and independent external probe.
P08	Expose only Caddy; use intended local or public TLS mode.	Config plus ss/external scan; avoid accidental wildcard local bind.
P09	Mount file-backed secrets only into services that need them.	Compose services/secrets mapping; service permissions review.
P10	Restrict host secret-file and backup permissions.	stat/ACL output archived without file contents.
P11	Verify credential rotation and reauthorization safely.	Rotation run record; old token rejected; new token permitted.

ID	Acceptance condition	Required proof / reviewer field
P12	Apply safe logging: no tokens, passwords, or sensitive payloads.	Representative structured logs and redacted error review.
P13	Keep container images and runtime environment separate from credentials.	Inspect image config/history; no passwords in image or repository.

Evidence log for this gate: control ID → PASS / FAIL / PENDING / N/A → non-secret artifact path or report ID → UTC timestamp → reviewer. Do not write passwords or tokens on this sheet.

Validate Compose policy without showing credentials

```
python review_h.py
python -m unittest test_review_h -v
# Both commands read YAML; neither reads secret contents
```

Inspect exact runtime settings after approved deployment

```
docker compose -f compose.prod28.yaml -f compose.local28.yaml ps
docker inspect --format '{{.Config.User}} | read-only={{.HostConfig.ReadonlyRootfs}}' CONTAINER_ID
# Read-only inspect; check effective process UID separately
```

Applicability note: the Chapter 28 API, publisher, worker, and migration image are hardened with read-only filesystems and dropped capabilities; vendor database and Caddy behavior require their own tested boundaries. The proxy intentionally retains NET_BIND_SERVICE.

H.3 Production Host, Published-Port Exposure, and Compose Release Review

Release question: is the specific Docker host safe to expose, provisioned for recovery, and running the exact release you reviewed?

Choose the TLS route deliberately. Local mode publishes only 127.0.0.1:8443; it uses Caddy's local CA and requires client trust. Public mode requires DNS control and an approved external 80/443 exposure. Do not run both overrides at once. Docker published ports can bypass simplistic ufw assumptions.

Migration/recovery order. Capture the last verified backup, approve the image digest, start DB and Redis, execute the one-off migration, then start the API and proxy. A previous image does not reverse a schema migration: require an explicit compatibility strategy.

ID	Acceptance condition	Required proof / reviewer field
D01	Approve host OS, Engine/Compose versions, patch age, and disk headroom.	Host inventory, patch timestamp, storage/capacity readings.
D02	Protect SSH, daemon access, and privileged deployment accounts.	SSH key policy, denied unauthenticated access, membership list.
D03	Check Docker publishing rules; do not assume ufw alone protects ports.	External scan on correct network and published-port review.
D04	Use only one TLS override for local or public deployment.	Validated merged config; local 127.0.0.1:8443 or approved 80/443.

ID	Acceptance condition	Required proof / reviewer field
D05	Use verified certificates and monitored renewal path.	Hostname/chain verified; renewal rehearsal and expiry monitoring.
D06	Verify deployment image and project identifiers are recorded.	Immutable digest, Compose project, versioned release manifest.
D07	Run migrations as controlled one-off work and test compatible rollback.	Migration run ID; backward-compatibility plan and backup point.
D08	Confirm DB/Redis persistent volumes and separate backups.	Volume inventory, backup checksum, independent restore evidence.
D09	Check health, logs, resource budgets, and restart behavior.	Healthy services, bounded restart counts, resource samples.
D10	Prove fail-safe deployment gates and post-deploy tests.	Approval workflow, smoke report, operator ownership and window.
D11	Record recovery point and successfully drilled recovery procedure.	Last tested restore ID, RPO/RTO observed, drill approver.

Evidence log for this gate: control ID → PASS / FAIL / PENDING / N/A → non-secret artifact path or report ID → UTC timestamp → reviewer. Do not write passwords or tokens on this sheet.

Check the intended merged configuration

```
docker compose -f compose.prod28.yaml -f compose.local28.yaml config --quiet
# Public host: use compose.public28.yaml instead of compose.local28.yaml
# Check live publishing with docker compose ... ps and sudo ss -lntp
```

Capture certificate and host evidence

```
openssl s_client -connect HOST:443 -servername HOST </dev/null
# Review certificate chain/expiry; do not paste private keys
# Probe externally from a separate trusted host/network
```

Host test boundary: compose config --quiet checks the merged model, not the live firewall or effective routes. Test published ports from a different machine or network; do not rely only on the same host's loopback test.

H.4 Authentication, Rate-Limit, and Acceptance Evidence Worksheet

Release question: does the deployed service enforce its access contract, withstand basic abuse, and have independently checked operational evidence?

Only live requests establish access control. Reading authorization code is not enough. Use the existing TLS-verified smoke utility, capture statuses and IDs, and run write probes only in isolated staging. The in-process mutation limiter is not a distributed or edge-level DDoS control.

Make the release decision auditable. Every control is PASS, FAIL, or N/A with justification. A missing test is PENDING, not PASS. The final sign-off must identify an approver, the exact digest, open exceptions, and immutable or controlled evidence references.

ID	Acceptance condition	Required proof / reviewer field

ID	Acceptance condition	Required proof / reviewer field
A01	Reject missing, invalid, and expired credentials.	Live HTTP 401 evidence; no secret values in captured logs.
A02	Enforce viewer read-only role for deployment resources.	Viewer GET allowed; POST/PATCH/DELETE denied with 403.
A03	Verify operator writes, audit fields, and data validation.	Staging-only mutation report with record ID and cleanup proof.
A04	Prove TLS hostname and CA trust, without insecure verification bypass.	Client certificate verification succeeds; wrong-hostname fails.
A05	Check unauthenticated health policy versus protected data endpoints.	/health and /ready contract; no sensitive data on public routes.
A06	Demonstrate expected 429 when write burst exceeds local limit.	Safe staging traffic report, Retry-After, recovery after window.
A07	Assess rate-limit limitations under multiple API replicas/restarts.	Record in-process scope; edge/global control plan if exposed publicly.
A08	Confirm correlation IDs, safe error responses, and log redaction.	Client request ID matches logs; no authorization header/body.
A09	Prove API-to-PostgreSQL and worker job lifecycle.	Read/write/persist and job completion/replay evidence.
A10	Prove backup restore AND app-image rollback separately.	Restore record counts; same records after image rollback.
A11	Attach evidence to release receipt and prohibit evidence-free sign-off.	Checklist status, artifact paths/hashes, named reviewer.
A12	Reject release with unresolved blockers; expire any approved exceptions.	HOLD or exception ID with scope, owner, expiry, mitigation.

Evidence log for this gate: control ID → PASS / FAIL / PENDING / N/A → non-secret artifact path or report ID → UTC timestamp → reviewer. Do not write passwords or tokens on this sheet.

Run the existing TLS-verified, non-mutating smoke check

```
python ops32/smoke32.py --url https://HOST --private-dir private
# Local CA only: add --ca-file /path/to/trusted-ca.crt
# --staging-write is allowed ONLY in disposable staging
```

Record test and recovery outcomes

```
python ops32/release_receipt.py verify --file evidence/capstone32/receipt.json
# Verify actual backup restoration and rollback as distinct tests
# Save exit status, UTC timestamp, approver, and evidence hash
```

Rate-limit note: DeployBoard’s window limiter is per Python process and is not sufficient as an internet-facing, distributed abuse-control system. Record edge controls if the application is made publicly accessible.

FINAL RELEASE DECISION

Complete the worksheet below only after the four gates are evaluated. A checked box is not evidence: write the report identifier, actual digest, reviewer, and result for each required test.

Gate	Outcome	Evidence location / ID	Reviewer + UTC date
H.1 gate	PENDING	_____	_____
H.2 gate	PENDING	_____	_____
H.3 gate	PENDING	_____	_____
H.4 gate	PENDING	_____	_____

Exact registry image@sha256 digest:

_____

Source revision / workflow run / SBOM and scan IDs:

_____

Backup point / restore drill / rollback evidence IDs:

_____

Approved risk exceptions with owner and expiry:

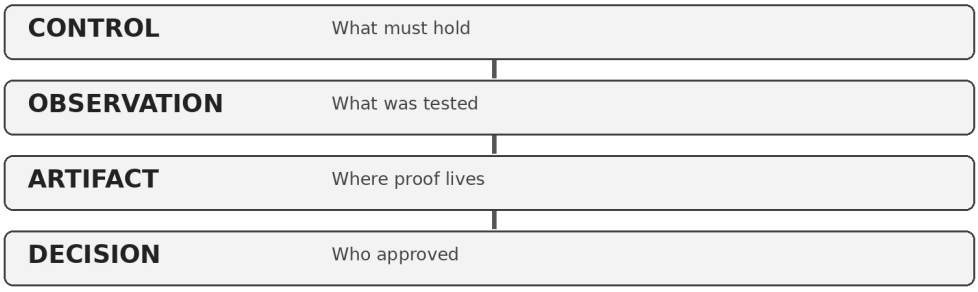
_____

Final decision (select one): HOLD / APPROVE LOCAL ONLY / APPROVE PUBLIC

Release owner / independent reviewer / UTC date:

_____

Mandatory release gate. All applicable controls must PASS with evidence. Any FAIL or mandatory PENDING blocks approval; N/A requires written rationale. Exceptions must be explicit, risk-accepted, time-limited, and consistent with the organization's policy. No program output can replace a named human release decision.



OFFLINE COMPOSE CHECK — COMPLETE SUPPORTING SOURCE

The small Python utility below checks selected declared Docker Compose boundaries from Chapter 28. It has no network access, never reads secret contents, and

deliberately finishes with “PENDING” for real-world tests. Copy review_h.py and test_review_h.py beside the three Compose YAML files before running. PyYAML is already pinned in the book’s requirements-ci.txt.

Run these three commands in order, from the folder containing the YAML files:

```
python -m pip install -r requirements-ci.txt
python -m unittest test_review_h -v
python review_h.py
```

review_h.py • complete source listing

```
"""DeployBoard Appendix H: conservative, offline Compose configuration checks.

Requires PyYAML from book-supplied requirements-ci.txt. Does not access Docker,
read credential contents, contact a registry, or authorize a deployment.
"""
from pathlib import Path
import sys
import yaml

ROOT = Path(__file__).resolve().parent
REQUIRED = {"db", "redis", "api", "publisher", "worker", "proxy", "migrate"}
APP = ("api", "publisher", "worker", "migrate")


def load(name):
    with (ROOT / name).open(encoding="utf-8") as stream:
        data = yaml.safe_load(stream)
    if not isinstance(data, dict) or not isinstance(data.get("services"), dict):
        raise ValueError(f"Invalid Compose structure: {name}")
    return data


def inspect(base, local, public):
    faults = []
    services = base.get("services", {})
    if not REQUIRED.issubset(services):
        faults.append("Missing required production services")
    if not base.get("networks", {}).get("private", {}).get("internal"):
        faults.append("Private network is not declared internal")
    for name in ("db", "redis", "api", "publisher", "worker", "migrate"):
        if "ports" in services.get(name, {}):
            faults.append(f"{name} publishes a host port")
    if "ports" in services.get("proxy", {}):
        faults.append("Base proxy publishes without a TLS-mode override")
    if "private" in services.get("proxy", {}).get("networks", []):
        faults.append("Proxy is attached to database-only network")
    for name in APP:
        svc = services.get(name, {})
        if "build" in svc:
            faults.append(f"{name} builds on production host")
        if svc.get("read_only") is not True:
            faults.append(f"{name} missing read-only root filesystem")
        if "ALL" not in svc.get("cap_drop", []):
            faults.append(f"{name} has not dropped all capabilities")
        if not any("no-new-privileges" in str(v) for v in svc.get("security_opt", [])):
            faults.append(f"{name} missing no-new-privileges")
```

```
            if not svc.get("secrets"):
                faults.append(f"{name} does not declare file-mounted secrets")
        if "tools" not in services.get("migrate", {}).get("profiles", []):
            faults.append("Migration is not profile-gated")
        expected = {"db_password", "viewer_token", "operator_token"}
        if not expected.issubset(base.get("secrets", {})):
            faults.append("Required file-backed secret declarations are absent")
        for key in expected:
            item = base.get("secrets", {}).get(key, {})
            if "file" not in item:
                faults.append(f"{key} is not sourced from a file")
        local_ports = local.get("services", {}).get("proxy", {}).get("ports", [])
        if local_ports != ["127.0.0.1:8443:443"]:
            faults.append("Local TLS proxy must publish only 127.0.0.1:8443")
        public_ports = public.get("services", {}).get("proxy", {}).get("ports", [])
        if sorted(str(v) for v in public_ports) != ["443:443", "80:80"]:
            faults.append("Public TLS proxy must publish only intended 80/443")
        return faults

    def main():
        try:
            faults = inspect(load("compose.prod28.yaml"),
                             load("compose.local28.yaml"),
                             load("compose.public28.yaml"))
        except (OSError, ValueError, yaml.YAMLError) as exc:
            print("HOLD: configuration unavailable or invalid:", type(exc).__name__)
            return 2
        for fault in faults:
            print("HOLD:", fault)
        if faults:
            return 2
        print("PASS: selected offline Compose safety checks")
        print("PENDING: live runtime, network, TLS, auth, backup, and CI evidence")
        print("This check is NOT a production release approval.")
        return 0

    if __name__ == "__main__":
        sys.exit(main())
```

test_review_h.py • complete source listing

```
    """No Docker Engine required. Use: python -m unittest test_review_h -v"""
    import copy
    import unittest
    import review_h

    class ComposeSecurityTests(unittest.TestCase):
        @classmethod
        def setUpClass(cls):
            cls.base = review_h.load("compose.prod28.yaml")
            cls.local = review_h.load("compose.local28.yaml")
            cls.public = review_h.load("compose.public28.yaml")

        def check(self, base=None, local=None, public=None):
            return review_h.inspect(base or self.base, local or self.local,
```

```
                                public or self.public)

    def test_supplied_files_meet_static_checks(self):
        self.assertEqual(self.check(), [])

    def test_exposed_database_is_rejected(self):
        data = copy.deepcopy(self.base)
        data["services"]["db"]["ports"] = ["5432:5432"]
        self.assertIn("db publishes a host port", self.check(base=data))

    def test_missing_no_new_privileges_is_rejected(self):
        data = copy.deepcopy(self.base)
        data["services"]["api"]["security_opt"] = []
        self.assertTrue(any("no-new-privileges" in x for x in self.check(base=data)))

    def test_local_wildcard_bind_is_rejected(self):
        data = copy.deepcopy(self.local)
        data["services"]["proxy"]["ports"] = ["8443:443"]
        self.assertTrue(any("Local TLS" in x for x in self.check(local=data)))

    def test_missing_secret_file_is_rejected(self):
        data = copy.deepcopy(self.base)
        data["secrets"]["db_password"] = {"environment": "DB_PASSWORD"}
        self.assertTrue(any("db_password" in x for x in self.check(base=data)))

if __name__ == "__main__":
    unittest.main()
```

Expected offline output (not a release approval): “PASS: selected offline Compose safety checks” followed by “PENDING: live runtime, network, TLS, auth, backup, and CI evidence.” Failed policy tests should report HOLD and exit with code 2.

Appendix I — Backup, Restore,Deployment, and Rollback Runbooks

A reference you can use during a planned change—or when the clock is running. These procedures operate on the DeployBoard single-host Compose release from Chapters 28–32. They deliberately separate an application image rollback from database restoration, because the former cannot undo a schema change or committed data.

Runbook	Use it when	Hard stop / first safeguard
I.1 · Backup and restore	Protect or prove PostgreSQL recoverability	Restore into a new project; never the live volume
I.2 · Restart / redeploy	Recover an app process or ship an approved digest	Record prior digest and schema revision first
I.3 · Rollback decision	Release fails after deployment	Stop if the old binary cannot read the new schema

Runbook	Use it when	Hard stop / first safeguard
I.4 · Evidence templates	Record operations and recovery drills	Keep tokens, passwords and private keys out of logs

OPERATING RULE These are Linux/bash host commands. In Windows, run them inside an appropriate Linux/WSL environment. The examples do not print secret values. Use a project copy and credentials you control; never perform the deliberate failure drill on production.

I.1 PostgreSQL Logical Backup and Restore Sequence

Objective. Produce a transactionally consistent PostgreSQL 17 custom-format archive, record its integrity metadata, and prove a restore into a completely separate database volume. An RPO is how much data you can afford to lose; an RTO is how quickly service must return. Neither is proven by a green backup job alone.

Before you begin	Proof to record
Confirm current production project is deployboard-prod28	docker compose -p deployboard-prod28 -f compose.prod28.yaml ps
Confirm digest-pinned DEPLOYBOARD_IMAGE and host space	Approved image reference; free bytes; UTC start time
Pause incompatible migrations and coordinate writes	Change ticket; backup window; application write plan
Confirm encrypted/off-host storage destination	Backup recipient, access controls, retention plan

RUN THIS SEQUENCE FROM THE DEPLOYBOARD PROJECT ROOT

```
umask 077
export DEPLOYBOARD_IMAGE='ghcr.io/OWNER/REPO@sha256:REPLACE_WITH_REAL_DIGEST'
# Replace the placeholder with the exact approved digest before proceeding.
bash tools/backup29.sh ./backups29
# Use the file actually printed by the preceding command.
python3 tools/archive29.py verify ./backups29/ACTUAL_FILE.dump
bash tools/restore29.sh ./backups29/ACTUAL_FILE.dump
# The restore script creates a fresh deployboard-recovery29-* project.
# Capture output and compare counts, sample IDs, and Alembic revision.
```

The backup includes one PostgreSQL database at one logical snapshot. It does not include Redis stream persistence, unacknowledged task state, TLS private keys, Caddy volumes, or external object storage. Preserve those components through their own tested procedures. A SHA-256 manifest detects accidental alteration relative to that manifest; it does not encrypt a backup or authenticate a hostile copy.

VERBATIM RECOVERY TOOLS SHIPPED IN THE DEPLOYBOARD CANONICAL KIT

The following four listings are reproduced from the Chapter 29 source and also appear in Appendix E. Create each file at the shown path, preserve LF line endings, and run the commands above. All file contents are present here; an external repository is not required.

COMPLETE FILE LISTING | TOOLS/BACKUP29.SH

```
#!/usr/bin/env bash
# Linux deployment host; from the DeployBoard project root.
set -euo pipefail
umask 077
cd "$(dirname "$0")/.."
: "${DEPLOYBOARD_IMAGE:?Export the deployed digest-pinned image first}"
[[ "$DEPLOYBOARD_IMAGE" == *@sha256:* ]] || { echo 'HOLD: digest image required' >&2; exit 2; }
DEST="${1:-./backups29}"
mkdir -p -- "$DEST"
DEST="$(cd "$DEST" && pwd -P)"
chmod 700 -- "$DEST"
DC=(docker compose -p deployboard-prod28 -f compose.prod28.yaml)
"${DC[@]}" ps -q db | grep -q . || { echo 'HOLD: production db is not running' >&2; exit 2; }
STAMP="$(date -u +%Y%m%dT%H%M%SZ)"
FINAL="$DEST/deployboard-$STAMP.dump"
[[ ! -e "$FINAL" ]] || { echo 'HOLD: name already exists' >&2; exit 2; }
TEMP="$(mktemp "$DEST/.backup29.XXXXXXXX")"
trap 'rm -f -- "$TEMP"' EXIT
# stdout is a binary PostgreSQL archive; never mix log messages into it.
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/db_password)"
  exec pg_dump -h 127.0.0.1 -U "$POSTGRES_USER" \
    -d "$POSTGRES_DB" -Fc -Z 6 --no-owner --no-acl
' > "$TEMP"
[[ -s "$TEMP" ]] || { echo 'HOLD: empty dump' >&2; exit 2; }
"${DC[@]}" exec -T db pg_restore --list < "$TEMP" > /dev/null
mv -- "$TEMP" "$FINAL"
python3 tools/archive29.py record "$FINAL"
echo "BACKUP RECORDED: $FINAL"
echo 'Transfer securely; a checksum alone is not encryption or proof of trust.'
```

COMPLETE FILE LISTING | TOOLS/ARCHIVE29.PY

```
"""Create and verify portable integrity metadata; NOT encryption or authenticity."""
from __future__ import annotations
import argparse
from datetime import datetime, timezone
import hashlib
import json
from pathlib import Path
import sys


def digest_file(path: Path) -> str:
    h = hashlib.sha256()
    with path.open('rb') as source:
        for chunk in iter(lambda: source.read(1024 * 1024), b''):
            h.update(chunk)
    return h.hexdigest()


def record(path: Path) -> Path:
    if not path.is_file() or path.stat().st_size < 100:
        raise ValueError('archive missing or suspiciously small')
    manifest = {
```

```
        'format': 'deployboard-pgdump-custom-v1',
        'archive': path.name,
        'bytes': path.stat().st_size,
        'sha256': digest_file(path),
        'created_utc': datetime.now(timezone.utc).isoformat(),
        'postgres_major': 17,
        'scope': 'one PostgreSQL database; Redis and TLS state excluded',
    }
    out = path.with_name(path.name + '.json')
    out.write_text(json.dumps(manifest, indent=2) + '\n', encoding='utf-8')
    out.chmod(0o600)
    return out


def verify(path: Path) -> dict:
    manifest = path.with_name(path.name + '.json')
    data = json.loads(manifest.read_text(encoding='utf-8'))
    if (data.get('format') != 'deployboard-pgdump-custom-v1'
            or data.get('archive') != path.name
            or data.get('bytes') != path.stat().st_size
            or data.get('postgres_major') != 17
            or data.get('sha256') != digest_file(path)):
        raise ValueError('backup metadata mismatch or archive tampering')
    return data


def main(argv=None) -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument('action', choices=['record', 'verify'])
    ap.add_argument('archive', type=Path)
    a = ap.parse_args(argv)
    try:
        result = record(a.archive) if a.action == 'record' else verify(a.archive)
        print('OK:', a.action, a.archive.name)
        return 0
    except (ValueError, OSError, json.JSONDecodeError) as exc:
        print('HOLD:', type(exc).__name__, file=sys.stderr)
        return 2


if __name__ == '__main__':
    sys.exit(main())
```

COMPLETE FILE LISTING | TOOLS/RESTORE29.SH

```
#!/usr/bin/env bash
# Restore ONLY into a new, private, independent recovery project.
set -euo pipefail
umask 077
cd "$(dirname "$0")/.."
[[ $# -ge 1 ]] || { echo 'Usage: bash tools/restore29.sh backup.dump [recovery-project]' >&2; exit 2; }
ARCHIVE="$(python3 -c 'from pathlib import Path; import sys; print(Path(sys.argv[1]).resolve())' "$1")"
PROJECT="${2:-deployboard-recovery29-$(date -u +%Y%m%dt%H%M%Sz)}"
[[ "$PROJECT" =~ ^deployboard-recovery29-[a-z0-9-]+$ ]] || { echo 'HOLD: invalid recovery project name'
>&2; exit 2; }
```

```
python3 tools/archive29.py verify "$ARCHIVE"
DC=(docker compose -p "$PROJECT" -f compose.recovery29.yaml)
if docker volume inspect "${PROJECT}_dbdata" >/dev/null 2>&1; then
  echo 'HOLD: destination volume exists; choose a fresh project' >&2
  exit 2
fi
mkdir -p private
chmod 700 private
PASSFILE=private/recovery29_db_password.txt
if [[ ! -f "$PASSFILE" ]]; then
  python3 - "$PASSFILE" <<'PYCODE'
from pathlib import Path
import secrets, sys
p = Path(sys.argv[1]); p.write_text(secrets.token_urlsafe(36) + '\n')
p.chmod(0o600)
PYCODE
fi
[[ -s "$PASSFILE" ]] || { echo 'HOLD: empty recovery password' >&2; exit 2; }
"${DC[@]}" up -d db
READY=0
for _ in {1..30}; do
  if "${DC[@]}" exec -T db pg_isready -U deployboard -d deployboard >/dev/null 2>&1; then
    READY=1; break
  fi
  sleep 2
done
[[ "$READY" -eq 1 ]] || { echo 'HOLD: recovery database not ready' >&2; exit 2; }
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/recovery29_password)"
  exec pg_restore -h 127.0.0.1 -U "$POSTGRES_USER" \
    -d "$POSTGRES_DB" --no-owner --no-acl \
    --exit-on-error --single-transaction
' < "$ARCHIVE"
echo "RECOVERY RESTORED: $PROJECT (not production)"
bash tools/verify29.sh "$PROJECT"
echo "NEXT: log measured recovery time and preserve this project for inspection."
```

COMPLETE FILE LISTING | TOOLS/VERIFY29.SH

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")/.."
[[ $# -eq 1 ]] || { echo 'Usage: bash tools/verify29.sh recovery-project' >&2; exit 2; }
PROJECT="$1"
[[ "$PROJECT" =~ ^deployboard-recovery29-[a-z0-9-]+$ ]] || { echo 'HOLD: invalid recovery project' >&2;
exit 2; }
DC=(docker compose -p "$PROJECT" -f compose.recovery29.yaml)
# SQL contains no user-supplied identifiers; no database port is published.
SQL="SELECT 'deployments', count(*) FROM deployments
UNION ALL SELECT 'status_events', count(*) FROM status_events
UNION ALL SELECT 'background_jobs', count(*) FROM background_jobs
UNION ALL SELECT 'missing_event_parent', count(*) FROM status_events e
  LEFT JOIN deployments d ON d.id=e.deployment_id WHERE d.id IS NULL
UNION ALL SELECT 'missing_job_parent', count(*) FROM background_jobs j
  LEFT JOIN deployments d ON d.id=j.deployment_id WHERE d.id IS NULL;"
```

```
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/recovery29_password)"
  exec psql -X -v ON_ERROR_STOP=1 -h 127.0.0.1 \
    -U "$POSTGRES_USER" -d "$POSTGRES_DB" -At -F "|" -c "$1"
' sh "$SQL"
"${DC[@]}" exec -T db sh -ec '
  export PGPASSWORD="$(cat /run/secrets/recovery29_password)"
  exec psql -X -v ON_ERROR_STOP=1 -h 127.0.0.1 \
    -U "$POSTGRES_USER" -d "$POSTGRES_DB" -At \
    -c "SELECT version_num FROM alembic_version"
'
echo 'CHECK: orphan counts must be zero; compare sample IDs and records with release evidence.'
```

OPERATING RULE Do not run docker compose down --volumes, docker volume prune, pg_restore --clean, or a database initialization command against the live project while investigating recovery. If backups contain production information, secure their storage, off-host transfer, retention, and disposal under your organization's policy.

I.2 Safe Application Restart and Redeployment Sequence

Use "restart" when the current container needs to be restarted without changing its image or Compose configuration. Use "recreate" when a verified release image or configuration changed. Both are distinct from restoring persistent data. A Compose restart does not re-read an edited image or apply changed Compose settings.

Stage	Operator action	Go / hold evidence
1 · Confirm	Record current image digest and Alembic revision	Old image remains pullable; backup verified
2 · Validate	Check Compose interpolation without printing secrets	docker compose config --quiet succeeds
3 · Prepare	Pull approved digest; confirm registry response	Manifest identity recorded
4 · Database	Apply only approved, backward-compatible migration	Revision and old-binary compatibility documented
5 · Replace app	Recreate API, publisher, worker—do not destroy DB volume	New container IDs and requested image
6 · Verify	HTTPS, 401/403 auth, read/write, job lifecycle, logs	Expected codes and persisted record ID

ROUTINE PROCESS RESTART (NO CHANGED IMAGE OR CONFIGURATION)

```
cd /path/to/deployboard
export DEPLOYBOARD_IMAGE='ghcr.io/OWNER/REPO@sha256:REAL_DIGEST'
DC=(docker compose -p deployboard-prod28   -f compose.prod28.yaml -f compose.local28.yaml)
"${DC[@]}" config --quiet
"${DC[@]}" ps
"${DC[@]}" restart api publisher worker
"${DC[@]}" ps
"${DC[@]}" logs --tail 60 api worker
```

In the optional public-TLS path, use compose.public28.yaml instead of compose.local28.yaml. Never combine both TLS overrides. Be alert that Compose may return successfully while the application remains unhealthy; verify application behavior separately.

APPROVED IMAGE REDEPLOYMENT (CONTROLLED CHANGE WINDOW)

```
# Save PREVIOUS_IMAGE outside your shell history or change record as needed.
PREVIOUS_IMAGE='ghcr.io/OWNER/REPO@sha256:PRIOR_REAL_DIGEST'
export DEPLOYBOARD_IMAGE='ghcr.io/OWNER/REPO@sha256:NEW_REAL_DIGEST'
DC=(docker compose -p deployboard-prod28   -f compose.prod28.yaml -f compose.local28.yaml)
"${DC[@]}" config --quiet
"${DC[@]}" pull api publisher worker
# Run only a migration reviewed as backward-compatible.
"${DC[@]}" --profile tools run --rm migrate
"${DC[@]}" up -d --no-deps --force-recreate api publisher worker
"${DC[@]}" ps
"${DC[@]}" logs --tail 60 api publisher worker
# HTTPS smoke and authenticated CRUD/job checks are still required.
```

OPERATING RULE A digest-pinned application image must be available before this command sequence can work; placeholders above are deliberately invalid. Do not silently replace real digests with :latest. If the database is not healthy, stop and diagnose rather than force-starting dependent services.

OPERATIONAL VERIFICATION

```
# Local CA path: export the Caddy root certificate as in Chapter 28.
curl --fail --show-error --cacert ./local-root-ca.crt   https://localhost:8443/ready
# Expected: HTTP success over verified TLS; not merely a proxy homepage.
python3 tools/smoke28.py --url https://localhost:8443   --ca-file ./local-root-ca.crt --private-dir ./private
# Capture exact exit code, sanitized results, and a known deployment ID.
```

The built-in smoke check verifies HTTP readiness and authorization. It does not replace a distinct background-job test, a data-persistence check, disk/CPU review, or a live external port scan. Record each result separately.

I.3 Release Rollback and Migration Decision Tree

Rollback is a release decision, not a reflex. Preserve the failed build’s digest, timestamps, container events, and relevant redacted logs first. Then evaluate whether the previous image supports the current database schema and whether in-flight jobs can replay safely.

Condition	Safe operator decision
Old image works with current schema; no incompatible writes	Pin old digest; recreate API, publisher and worker; verify records and jobs
Schema expanded compatibly, but candidate service failed	Roll back app tier; keep schema; preserve new fields for later remediation
Destructive schema migration or unknown compatibility	HOLD: do not start old image; escalate to planned forward fix or data recovery

Condition	Safe operator decision
Evidence indicates logical data corruption	Isolate affected writers; plan separate recovery with approved RPO/RTO
Backup is old; restoring would lose accepted writes	Obtain data-owner approval and communicate explicitly before restore

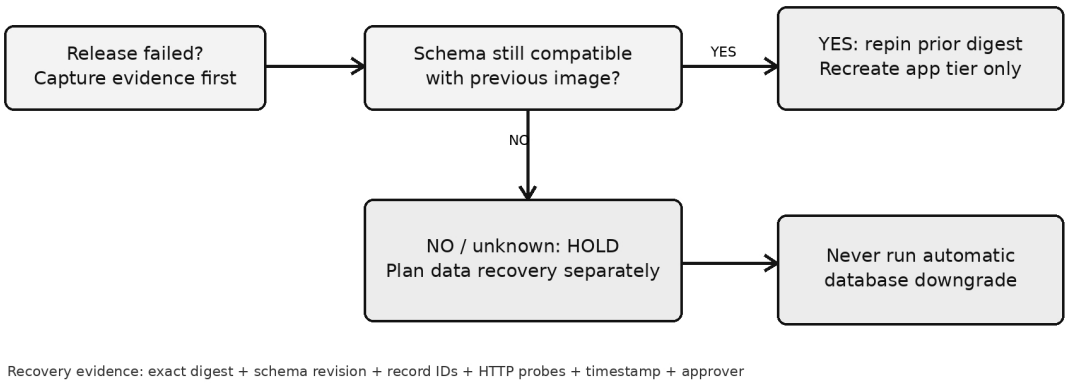


Figure I.1 | Application rollback decision—database state is a separate concern.

SAFE STAGING FAILURE DRILL (NEVER PRODUCTION)

```
# Stage secrets live only under private/stage30/.
export DEPLOYBOARD_IMAGE='ghcr.io/OWNER/REPO@sha256:STAGING_REAL_DIGEST'
STAGE=(docker compose -p deployboard-stage30   -f compose.prod28.yaml -f compose.stage30.yaml)
"${STAGE[@]}" config --quiet
"${STAGE[@]}" up -d db redis
"${STAGE[@]}" --profile tools run --rm migrate
"${STAGE[@]}" up -d api publisher worker proxy
# Deliberately break ONLY the staging API startup (exit 42).
docker compose -p deployboard-stage30   -f compose.prod28.yaml -f compose.stage30.yaml   -f
compose.fault30.yaml up -d --no-deps api
"${STAGE[@]}" ps api
"${STAGE[@]}" logs --tail 40 api
# Remove fault override and recreate the normal staging API.
"${STAGE[@]}" up -d --no-deps --force-recreate api
"${STAGE[@]}" ps api
# Re-run stage HTTPS, access-control, record-ID and worker acceptance.
```

For an actual application-only rollback, re-export DEPLOYBOARD_IMAGE with the previously approved digest and run the same --no-deps --force-recreate for the application tier after verifying compatibility. Do not run Alembic downgrade automatically. The production database volume is not replaced, and rolling back the image cannot undo writes already committed.

WHEN RESTORATION BECOMES A SEPARATE INCIDENT

Start a new recovery ticket with the approved snapshot, backup SHA-256, data owner, expected data loss, recovery environment, and change freeze. Exercise the restore in the isolated Chapter 29 project before choosing any production cutover method. Database promotion, reconciliation of later writes, and credential rotation require independent approval; they are deliberately not automated in this appendix.

I.4 Incident Log and Recovery Drill Templates

Evidence is not a screenshot of a green command. It identifies the exact change and proves the important invariants—same records, authorized behavior, healthy queue, private database networking, and recoverability—at a known time. Copy the templates below into your operations notebook or use the included text files.

INCIDENT TIMELINE TEMPLATE — NO CREDENTIALS

```
INCIDENT ID:                  UTC OPENED:
OPERATOR / REVIEWER:          SEVERITY / IMPACT:
AFFECTED HOST + COMPOSE PROJECT:
LAST HEALTHY APP IMAGE (DIGEST):
CURRENT APP IMAGE (DIGEST):     ALEMBIC REVISION:
SYMPTOM / USER-FACING IMPACT:
UTC TIME | OBSERVATION | COMMAND/LOG REF | ACTION | RESULT
      |          |             |       |
      |          |             |       |
DATA LOSS SUSPECTED? YES / NO / UNKNOWN
DO NOT INCLUDE passwords, tokens, key material or personal records.
DECISION: MONITOR / ROLLBACK APP / RESTORE PLAN / ESCALATE
APPROVER:                     UTC CLOSED:
FOLLOW-UP PREVENTION OWNER / DATE:
```

RECOVERY DRILL EVIDENCE TEMPLATE

```
DRILL ID:                     UTC START / UTC END:
PRODUCTION PROJECT: deployboard-prod28
ISOLATED RECOVERY PROJECT: deployboard-recovery29-...
BACKUP ARCHIVE / SHA-256 MANIFEST:
AGE OF ARCHIVE AT RESTORE (OBSERVED RPO):
TIME TO VERIFIED RESTORE (OBSERVED RTO):
DATABASE: PostgreSQL 17     ALEMBIC REVISION:
TABLES: deployments=__  status_events=__ background_jobs=__
ORPHAN EVENTS=__  ORPHAN JOBS=__  EXPECTED: ZERO
SAMPLE DEPLOYMENT IDS VERIFIED:
REDIS QUEUE STATE: SEPARATE RECOVERY PLAN / NOT RESTORED
CADDY/TLS STATE: SEPARATE PLAN / NOT RESTORED
OFF-HOST COPY + ENCRYPTION EVIDENCE:
RESULT: PASS / FAIL / INCOMPLETE    REVIEWER:
OPEN ACTIONS / NEXT DRILL DATE:
```

CHANGE-WINDOW / RELEASE DECISION SHEET

Required evidence	Record
Old and new app image digests	_____
Backup name/hash and proven restore date	_____
Schema compatibility / forward migration approval	_____
TLS + 401 + 403 + authorized CRUD + worker outcome	_____
Persistence and private-port exposure checks	_____
Approver; GO / HOLD / ROLLBACK; UTC timestamp	_____

Acceptance threshold. A recovery drill is APPROVED only when the archive matches its integrity metadata, pg_restore completes with zero reported errors, orphan relationships are zero, representative records match the evidence sheet, and observed RPO/RTO are within the organization’s agreed objectives. Any unknown criterion is HOLD, not PASS.

OPERATOR QUICK INDEX

Need	Command / location
Backup production PostgreSQL	bash tools/backup29.sh ./backups29
Verify custom archive metadata	python3 tools/archive29.py verify FILE.dump
Restore and verify in isolation	bash tools/restore29.sh FILE.dump
Review container state and logs	docker compose -p deployboard-prod28 -f compose.prod28.yaml ps
Staging rollback rehearsal	compose.stage30.yaml plus compose.fault30.yaml
Full canonical source / fixtures	Appendix E; current source in project root

REHEARSAL VARIATIONS: VERIFY THE RUNBOOK UNDER STRESS

Rehearsal	What must remain true
Old archive fails hash verification	Stop before restoring; locate a verified, trusted copy
Healthy DB, API restart only	Persistent records and Alembic revision remain unchanged
Candidate API fails to start	Old image rollback occurs only after schema compatibility check
Redis unavailable during restore	Do not equate restored SQL rows with completed background work
TLS issuer or registry unavailable	Use preapproved local artifacts and recovery paths; record degraded service

Rehearse the procedure on a nonproduction host at an interval suited to your recovery objectives. Compare actual recovery time to the target; include time to discover the failure, locate the archive, transfer it, restore PostgreSQL, validate relationships, renew credentials where appropriate, and reopen application writes. A fast database import is not the same as an acceptable service RTO.

On every drill, verify at least one record created before the simulated outage and one workflow outcome after service is restored. Check that the final recovery project has its own network and named volume, and retain a sanitized hash inventory of the evidence. If the evidence is incomplete, log the result as INCOMPLETE and assign a retest; do not turn the result into an approval by inference.

Appendix J — Glossary, Version Notes, and Official References

A reliable container deployment begins with precise vocabulary, a reproducible version record, and sources you can verify.

Use this closing reference alongside the DeployBoard source listing in Appendix E and the release, security, and recovery runbooks in Appendices H–I. Terms are

intentionally brief; whenever a capability depends on the host platform, Docker release, container image, or Compose plugin, verify the installed behavior rather than assuming the book has certified it.

REFERENCE RULE A pinned file is not a tested deployment. An offline test is not a cross-platform test. Keep source versions, runtime evidence, and release approval in separate records.

J.1 Container, Image, Registry, Compose, and OCI Glossary

The following definitions use Docker and OCI terminology as documented in [D01–D11] and [O01–O03]. The last column ties each term to the DeployBoard project so this appendix remains a working reference rather than a generic word list.

Containers, images, and distribution

Term	Meaning	DeployBoard cue
Container	An isolated running process created from an image, with its own writable layer.	The api or worker instance
Image	A read-only layered package containing a filesystem and runtime metadata.	DeployBoard FastAPI image
Image layer	A filesystem change set reused across image builds and downloads.	Python base-image layers
Dockerfile	Declarative build instructions used to construct an image.	Root Dockerfile
Build context	Files made available to a build; limit using .dockerignore.	DeployBoard project root
Image tag	Human-readable mutable reference to an image, not an immutable guarantee.	postgres:17.6-alpine
Digest	Content-addressed identifier such as sha256:… for a manifest.	Release image freeze
Registry	Service storing and distributing image manifests and blobs.	Source of pulled images
Repository	Named collection of related images within a registry.	library/postgres
Manifest	Metadata describing an image and its content descriptors.	OCI image manifest
Manifest list / image index	Descriptor for one or more platform-specific images.	amd64 vs arm64 selection
OCI image spec	Standard for image layout, descriptors, configuration, and layers.	Portable image format
OCI distribution spec	Standard API for registry pull, push, and content discovery.	Registry interoperability
OCI runtime spec	Standard for container runtime execution bundles and processes.	Low-level runtime contract

Build, networking, and storage

Term	Meaning	DeployBoard cue
Docker Engine	Daemon and API responsible for local containers, images, networks, and volumes.	The running host daemon
Docker CLI	Client invoking Engine/Buildx/Compose commands; versioned separately.	docker version
BuildKit	Backend that resolves Dockerfile build graphs and build cache.	Image build
Buildx	Docker CLI plugin exposing BuildKit builders and advanced outputs.	docker buildx version
Build cache	Reusable results of prior build steps; improves speed, not correctness proof.	Dependency-layer reuse
Named volume	Docker-managed persistent data independent of container lifetime.	db_data for PostgreSQL
Bind mount	Host path exposed to a container; coupled to host permissions and filesystem.	nginx/default.conf
Writable layer	Container-specific changes discarded when its container is removed.	Not database durability
Bridge network	Host-local container network for service-to-service connectivity.	backend network
Published port	Host-side listener forwarding traffic to a container port.	127.0.0.1:8088
Service DNS	Compose service name resolved within the project network.	db, redis, api
Health check	Configured test producing health status; not always business readiness.	pg_isready, /healthz
Readiness endpoint	Application endpoint showing whether dependencies permit useful work.	GET /readyz
Secret mount	File made available to an authorized service, not a plaintext env substitute.	db_password at /run/secrets

Compose, release, and operational evidence

Term	Meaning	DeployBoard cue
Compose project	Named group of services and resources from a Compose configuration.	name: deployboard
Compose service	Desired definition of a set of containers with one responsibility.	db, redis, migrate, api, worker, proxy
Compose plugin	Docker CLI extension using docker compose (space).	docker compose version
Compose override	Additional Compose file merged with the base configuration.	compose.dev.yaml
Profile	Opt-in grouping of services within Compose.	Optional launch grouping
Interpolation	Substitution of variables into the Compose model before execution.	${HTTP_PORT:-8088}
env_file	Service option that imports environment variables into a container.	Non-secret settings only
depends_on condition	Startup dependency criterion, optionally health/completion aware.	migrate waits for db
Restart policy	Engine behavior after container exit; not a root-cause fix.	unless-stopped
Migration	Versioned change to database schema or data structures.	001_core.sql → 002_indexes.sql
Idempotency key	Request identity allowing safe duplicate detection.	POST /v1/jobs/report
SBOM	Inventory of software components; separate from provenance and scanning.	Release supply-chain evidence
Provenance	Metadata about build source and process; verify provenance trust.	Build attestations
RPO / RTO	Maximum acceptable data loss / time to recover.	Restore drill objectives

DO NOT CONFUSE `EXPOSE 8080` documents a container port; it does not publish it. A `healthy` container is not proof of correct authorization. A successful `docker compose config` is not a successful deployment.

J.2 Tested-Version and Cross-Platform Compatibility Matrix

This section preserves the source freeze established in Appendix E (dated 20 October 2026) while recording what was actually verified. “Reference pinned” means a manifest, lock file, Dockerfile, or sample Compose file specifies that version. “Offline verified” refers to the Python/API and structural tests described in Appendix E; it does not imply containers ran in Docker.

DeployBoard reference freeze — 20 October 2026

Component	Book reference	What is fixed	Evidence still needed
Python base image	python:3.12.11-slim-bookworm	Pinned in Dockerfile	Image digest + container launch: NOT RUN
FastAPI	0.128.2	requirements.txt == pin	API behavior exercised offline
Uvicorn	0.48.0	requirements.txt == pin	Container HTTP stack: NOT RUN
SQLAlchemy	2.0.50	requirements.txt == pin	Offline SQLite test exercised
psycopg[binary]	3.2.10	requirements.txt == pin	Live PostgreSQL connection: NOT RUN
Redis client	6.4.0	requirements.txt == pin	Live Redis integration: NOT RUN
PostgreSQL image	postgres:17.6-alpine	Compose tag	Live database / restore: NOT RUN
Redis image	redis:7.4.6-alpine	Compose tag	Live broker / health: NOT RUN
Nginx image	nginx:1.28.0-alpine	Compose tag	TLS and proxy release: NOT RUN
Compose plugin	2.20+ project baseline	Appendix E manifest	Installed version: NOT CAPTURED
Docker Engine / Buildx	No universal version pinned	Capture on each host	Cross-platform live matrix: NOT RUN

Appendix E also pins pytest 9.0.2, httpx 0.28.1, and PyYAML 6.0.3 in requirements-dev.txt. Those are test-environment dependencies; they are not proof of a live Engine, image-pull, or cluster deployment.

IMAGE FREEZE Tags are references, not immutable release artifacts. Resolve and retain a platform-specific sha256 digest during the final live release check. Do not invent a digest from a documentation example.

Cross-platform capability and verification scope

Environment	Capability boundary	Owner-run evidence
Windows + Docker Desktop (Linux containers)	WSL 2–based or supported alternative backend; run Docker CLI in intended host/WSL context.	Confirm virtualization, docker context, Compose config, bind paths, port 8088, owner ACLs.
macOS + Docker Desktop	Supported macOS release and hardware; container workloads use a Linux VM.	Confirm host architecture, Linux image availability, bind mounts, port, file sharing, tests.
Linux + Docker Engine / Compose plugin	Native daemon, host permissions, and kernel capabilities vary by distribution.	Check daemon, plugin, group/rootless permissions, volume, network and restore.
Linux + Docker Desktop	Desktop runs its own VM and docker context; separate images/volumes from host Engine.	Verify docker context show before inspecting DeployBoard state.
Remote VPS / CI runner	Kernel, firewall, registry, secrets, CPU and architecture differ from laptop.	Capture exact version/build, image digests, acceptance and rollback evidence.

DeployBoard's `proxy` example is loopback-bound for a local pilot; a public deployment also requires independently verified TLS termination, authentication, rate

limiting, ingress policy, and backup/recovery evidence (Appendices H–I). Do not mark “public release ready” solely because development tests passed.

Record actual versions without revealing credentials

From a terminal in the DeployBoard project directory, use these read-only checks. Do not upload full `docker info`, verbose Compose renderings, tokens, `.env` files, or secret mounts to public issue trackers. `docker compose config --quiet` parses the Compose model without printing substituted values.

```
docker version
docker compose version
docker buildx version
docker context show
docker info --format '{{.OSType}} / {{.Architecture}}'
python --version
# After creating your private .env and secrets:
docker compose -f compose.yaml -f compose.dev.yaml config --quiet
docker compose config --services
```

These commands use the same spelling in PowerShell, Bash, and macOS terminals. On Windows, `python` may be `py -3`; record the executable actually used. Missing Compose configuration files or secrets make validation a NOT RUN result, not an automatic pass.

Reusable operator evidence receipt

RELEASE ID / timestamp (UTC): _____
OS / architecture / Docker context: _____
Docker client / Engine / API / Compose / Buildx: _____
Python / requirements lock hash: _____
Image references and resolved sha256 digests: _____
compose config: PASS / FAIL / NOT RUN
Offline tests: PASS / FAIL / NOT RUN
Live API + authorization: PASS / FAIL / NOT RUN
Database backup + isolated restore: PASS / FAIL / NOT RUN
TLS + rate limit (public only): PASS / FAIL / NOT RUN
Approved scope: LOCAL PILOT / PUBLIC / HOLD
Owner / date / evidence directory: _____

J.3 Official Docker, Compose, BuildKit, and OCI Documentation

The links below are primary sources, not search-result summaries. They were reviewed for this edition on 20 October 2026. Online interfaces and support ranges can change; use the installed CLI `--help`, the release notes, and your stored version receipt when a statement conflicts with the working environment.

Docker fundamentals and running services

D01 Docker — What is a container? — Processes and container lifecycle.
https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/

D02 Docker — What is an image? — Image layers, identities, and immutability.

https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/

D03 Docker — What is a registry? — Registries, repositories, and tags.
https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-registry/

D04 Docker — Volumes — Persistent database storage.
https://docs.docker.com/engine/storage/volumes/

D05 Docker — Bind mounts — Host-file and permission boundaries.
https://docs.docker.com/engine/storage/bind-mounts/

D06 Docker — Docker version — Client, daemon, and API version inspection.
https://docs.docker.com/reference/cli/docker/version/

D07 Docker — Docker Engine installation — Supported Linux installation routes.
https://docs.docker.com/engine/install/

Compose and build systems

D08 Docker — Compose Specification — Services, mounts, health and schema reference.
https://docs.docker.com/reference/compose-file/

D09 Docker — Version/name top-level elements — Obsolete version field; project naming.
https://docs.docker.com/reference/compose-file/version-and-name/

D10 Docker — Compose startup order — Depends-on health and completion conditions.
https://docs.docker.com/compose/how-tos/startup-order/

D11 Docker — Environment precedence — Environment entry priority.
https://docs.docker.com/compose/how-tos/environment-variables/envvars-precedence/

D12 Docker — Interpolation — Variables and missing-value semantics.
https://docs.docker.com/compose/how-tos/environment-variables/variable-interpolation/

D13 Docker — Compose config command — Merged-model validation.
https://docs.docker.com/reference/cli/docker/compose/config/

D14 Docker — BuildKit — Build backend and cache.
https://docs.docker.com/build/buildkit/

D15 Docker — Buildx build — Multi-platform builds, SBOM/provenance.
https://docs.docker.com/reference/cli/docker/buildx/build/

Platforms and Open Container Initiative

D16 Docker — Install Desktop on Windows — Backend and host requirements.
https://docs.docker.com/desktop/setup/install/windows-install/

D17 Docker — Install Desktop on macOS — OS and architecture support.
https://docs.docker.com/desktop/setup/install/mac-install/

D18 Docker — Install Desktop on Linux — Separate VM and docker context.
https://docs.docker.com/desktop/setup/install/linux/

O01 OCI — Specifications overview — Image, runtime, and distribution standards.
https://opencontainers.org/

O02 OCI — Image Specification — Format and descriptors.
https://github.com/opencontainers/image-spec

O03 OCI — Runtime Specification — Container runtime contract.
https://github.com/opencontainers/runtime-spec

O04 OCI — Distribution Specification — Registry API interoperability.
https://github.com/opencontainers/distribution-spec

J.4 Further Study and Platform Capability Boundaries

Work outward from DeployBoard's verified local result. Each stage has a distinct proof, and the next stage should not silently inherit the previous stage's approval.

A progressive path after the book

Next study area	What to learn	Proof to collect
Image integrity	Immutable digests, build provenance, dependency updates.	SBOM, digest lock, reviewed vulnerability findings.
Networks and identity	External reverse proxy, TLS lifecycle, service-to-service access, rate limits.	Negative authentication tests, certificate and ingress evidence.
Data recovery	PITR concepts beyond logical dumps, migration compatibility, RPO/RTO.	Restoration on a disposable target with measured duration.
Build systems	BuildKit caching, multi-platform manifests, CI runner isolation.	Rebuilt digest/metadata, platform matrix, provenance.
Capacity and scale	Resource constraints, observability, worker concurrency, queue fairness.	Load profile, saturation thresholds, alert receipts.
Orchestration beyond Compose	When single-host Compose no longer matches recovery and scale requirements.	Written migration rationale and tested deployment target.

Capability boundaries worth stating explicitly

Compose is not a full cluster scheduler. A successful single-host Compose setup does not provide multi-node orchestration or automatic cross-host failover.

OCI conformance is not application security. The image/runtime/distribution standards improve interoperability; they do not authenticate your DeployBoard endpoints or approve its secrets.

The host kernel and Docker context matter. Linux container features, filesystem behavior, architectures, and port access vary among native Engine and Desktop VMs.

A readiness endpoint is not a security review. `/readyz` can show database reachability while TLS, credential rotation, or policy controls remain missing.

Offline tests are narrower than production evidence. The Appendix E HTTP/API tests do not exercise live PostgreSQL, Redis, Docker Engine, Compose, proxy TLS, or restore drills.

Before marking the book's final release receipt complete

- □ Record every installed runtime version and the selected Docker context.
- □ Validate both intended Compose file combinations without publishing resolved secrets.
- □ Resolve image references to sha256 digests for each release architecture.
- □ Run the DeployBoard offline test suite and preserve a dated result.
- □ Run live API, migration, health, authorization, backup, and isolated restore checks.
- □ For public exposure, add and test the TLS edge, rate limiting, and ingress controls.
- □ Store the resulting version receipt, decision, and recovery evidence outside the running container.

When the installed platform differs from the book

Treat a newer version as a change requiring comparison, not as evidence that the example is obsolete. Capture the exact component that changed; check its official release notes; validate the fully merged Compose model; then rerun only the

acceptance checks that could be affected. Preserve the previous release receipt for comparison.

Observed difference	First safe check	Do not conclude
Compose plugin has changed	docker compose version; docker compose config --quiet	A valid model proves running containers are healthy.
Image tag resolves differently	Record digest and image architecture; inspect release notes	A familiar tag guarantees identical bytes.
Host changes between Desktop and Engine	docker context show; docker info; review mount paths	Same CLI means same images, volumes or network.
Database major or migration changed	Review forward/backward schema plan; stage restore	Rolling the image back reverses the schema.
Platform-specific build fails	Check buildx platforms, base-image availability, build logs	Source code is inherently non-portable.

Documentation update record — keep with your release receipt

CHECKED ON (UTC): _____
Component / old -> new version: _____
Official source URL and release-note date: _____
Relevant Compose/Dockerfile change: _____
Affected test IDs and observed results: _____
Rollback or migration decision: _____
Release status: APPROVED / HOLD / NOT RUN
Owner and retained evidence path: _____

A container that starts is a promising beginning. A system you understand, protect, and can recover is the result that matters.

You began with the question of how to run a container. Across this book, that question became a much more useful one: how do you build an application whose image is repeatable, whose connections are intentional, whose data survives replacement, and whose behavior can be verified?

DeployBoard carried that question from a single process to a working application with an API, PostgreSQL, background work, a reverse proxy, versioned images, tests, and operational evidence. Each new component introduced not just capability, but a responsibility: a new interface to secure, a dependency to observe, or a failure mode to rehearse.

The important change is not that you know more commands. It is that you can ask for proof before you trust a release—and collect the right evidence when something goes wrong.

The habits that make a deployment dependable

BUILD from deliberate inputs. Keep the source complete, the build context controlled, dependencies recorded, and the image identifiable by its digest. Reproducibility is a design choice, not an accident.

VERIFY the behavior that users and operators rely on. A green container status alone does not prove a database is reachable, an API enforces permissions, or a queued job eventually completes.

BREAK with boundaries. Practice failures using disposable data, isolated project names, and an explicit cleanup plan. A controlled experiment makes real incidents less surprising.

REPAIR with evidence. Read the state, logs, mounts, networks, resource pressure, and recent changes before reaching for destructive commands. A restart is a test, not always a diagnosis.

SHIP with an owner and a recovery path. Know what is exposed, where credentials live, how backups are verified, and what may—or may not—be rolled back when the schema changes.

An operator's definition of done

Before calling the next DeployBoard release ready, keep a short evidence record that answers five questions:

THE QUESTION	EVIDENCE TO RETAIN
What is running?	Release image digest, Compose configuration, platform versions, and deployment date.
Does it work?	API and worker checks, authorization denials, readiness checks, and representative user flows.
What can reach it?	Published-port audit, host firewall evidence, TLS configuration, and internal-network boundaries.
What survives failure?	Backup identity, storage inventory, isolated restore result, and measured RPO/RTO.
Who takes action?	On-call owner, incident notes, change approval, and safe repair or rollback procedure.

Know the boundary of this architecture

The DeployBoard reference project teaches secure, reproducible single-host Docker operation; it is not a blanket production certification. An internet-facing release requires reviewed TLS, authentication, exposure limits, dependency and image scanning, restore evidence, and controls appropriate to its actual threat model. A successful offline test is not proof that those controls were tested on the destination host.

Compose on one host can be a sound choice when its recovery time, availability, operational complexity, and cost meet your requirements. If they do not, it is time to study orchestration and platform services—but carry the same habits with you. More infrastructure will not replace a clear contract, an observed system, or a practiced restore.

What you can take into the next project

You can now choose a base image critically, package an application with a reproducible Dockerfile, connect services privately, make durable storage intentional, automate tests, publish a traceable image, and deploy with a stop condition. Just as importantly, you can tell the difference between a container that is running and a system that is working.

Keep improving the system you actually operate. Preserve the evidence, rehearse the recovery, and leave the next person a safer place to begin.

A final exercise for the next release

Choose one modest application you actually use. Before changing it, write down its service boundaries, published ports, credentials, persistent data, and the exact image you intend to run. Capture a baseline request and the logs that explain the normal path. Then make one controlled change, validate the resulting behavior, and record what you would do if the change failed.

After that, practice the least glamorous step: restore a backup into an isolated environment and prove the recovered data is useful. If the recovery fails, the exercise has succeeded in finding something you can repair while there is still time. Repeat the drill after meaningful database or infrastructure changes.

Keep a concise release record for yourself or the next operator. Good records are not bureaucracy when a service is down; they are a shortcut back to reliable decisions. The goal is not perfection. It is the ability to notice, explain, and recover from the ordinary failures that real software eventually meets.

You can build what you ship. You can verify it, recover it, and operate it with

A Request for Your Review

If this book helped you understand Docker, build a working DeployBoard application, or feel better prepared to operate a release, I would value hearing about your experience.

An honest review on the bookstore where you purchased Docker Projects Handbook can help other readers decide whether its practical, project-led approach is useful for them. Your review does not need to be long—or positive. What matters most is that it reflects what you actually experienced.

You might mention the idea that finally clicked, the exercise that helped you solve a real problem, or something you wish had been explained differently. Specific feedback is useful to future readers and to the improvement of future editions.

When the option is available, open the book's store listing and select the review feature. Reviews are entirely voluntary, and nothing is offered in exchange for a review.

Thank you for the time you invested in building, testing, troubleshooting, and recovering a system alongside this book. I hope those skills continue to pay off on projects of your own.

Auren Teklor
