

Docker Intermediate

Visual handwritten study guide - Tier 2

Understand

Clear, compact explanations

Visualize

Diagrams that show the moving parts

Practice

Readable commands and code

Remember

Highlighted revision points

65 topics - each topic begins on a fresh page
A digital asset by Sooraj | soorajdev.cloud

Course Map

Scan the complete learning path before you begin.

Part 1 - Engine and Environment

1	Welcome to Docker Intermediate	2	Docker Engine 29 in Practice
3	Configuring the Daemon with daemon.json	4	Contexts and Remote Daemons
5	Rootless Docker and User Namespaces	6	BuildKit Architecture and Dockerfile Syntax

Part 2 - Advanced Image Building

7	Cache Mounts and Bind Mounts in RUN	8	Build Secrets and SSH Forwarding
9	Build Cache Backends	10	Multi-Platform Images
11	buildx bake: Declarative Builds	12	Recipes: Node.js and Python Images
13	Recipes: Go, Java and Rust Images	14	Slimming and Inspecting Images

Part 3 - Registries and Supply Chain

15	Running a Private Registry	16	Harbor and Managed Registries
17	Tagging, Digests and Promotion Strategy	18	Vulnerability Scanning
19	SBOMs: Know What Is Inside	20	Signing Images with Cosign
21	Provenance and SLSA	22	Base Images and Update Policy

Part 4 - Networking and Storage

23	Bridge Networks Under the Hood	24	Port Publishing and the Firewall
25	macvlan, ipvlan and Host Networking	26	Overlay Networks and Multi-Host Basics
27	DNS, IPv6 and Proxy Settings	28	Storage Drivers and the Union Filesystem
29	Volumes, Drivers and Network Storage	30	Running Databases in Containers
31	Backup, Restore and Disaster Recovery		

Part 5 - Compose Mastery

32	The Compose Specification in Depth	33	Multiple Files, Merging and extends
34	include, Profiles and Large Projects	35	Environment, Interpolation and Secrets
36	Health Checks, Ordering and Resilience	37	Compose Watch and the Inner Loop
38	Testing with Compose and Testcontainers	39	Compose in Production
40	Reverse Proxies: Caddy, Traefik and nginx		

Part 6 - Security and Runtime

41	The Container Threat Model	42	Linux Capabilities
43	seccomp, AppArmor and SELinux	44	Secrets and Sensitive Data at Runtime
45	Protecting the Docker Daemon and Socket	46	Sandboxed Runtimes: gVisor and Kata
47	A Hardened Container, Step by Step	48	Audit, Monitoring and Incident Response

Part 7 - Operations

49	Logging Drivers and Log Pipelines	50	Metrics: cAdvisor, Prometheus and Grafana
51	Resource Limits and cgroups v2	52	Performance Tuning
53	Debugging Containers	54	Docker Swarm Essentials
55	Docker in CI/CD Pipelines	56	The Docker Engine API and SDKs
57	Docker Desktop, Model Runner and the Wider Toolbox		

Part 8 - Capstone Projects and Reference

PROJECT 1	Project 1: A Private Registry with a Signed Pipeline	PROJECT 2	Project 2: An Observability Stack
PROJECT 3	Project 3: A Hardened Production Deployment	61	Tier 2 Review Questions
APPENDIX A	Dockerfile and Image Checklist	APPENDIX B	Command Cheat Sheet
APPENDIX C	Troubleshooting Matrix	APPENDIX D	Glossary

Welcome to Docker Intermediate

What this tier assumes, what it adds, a quick self-check of the beginner material and the lab environment you will use throughout.

This is **Tier 2** of the Docker series. The beginner notes taught you to run containers, write Dockerfiles, persist data, connect containers and describe a stack in Compose. Here we make that knowledge **production-shaped**: faster and safer builds, trustworthy images, real networking and storage decisions, mature Compose setups, security hardening and the operational skills that keep containers healthy.

If you can answer the ten self-check questions below without looking anything up, you are ready. If not, spend a week with the Beginner notes first; this tier moves quickly.

What this tier covers

Part	Focus	You will be able to...
1. Engine and environment	Docker Engine 29, daemon configuration, contexts, rootless	Configure and operate the engine deliberately
2. Advanced image building	BuildKit, caching, bake, multi-platform, language recipes	Build small, fast, reproducible images
3. Registries and supply chain	Private registries, signing, SBOMs, scanning	Ship images you can prove and trust
4. Networking and storage	Bridge internals, macvlan, overlay, drivers, databases	Reason about packets and bytes
5. Compose mastery	Merging, include, profiles, watch, testing, production	Run real multi-service systems
6. Security and runtime	Capabilities, seccomp, secrets, isolation runtimes	Harden containers and the daemon
7. Operations	Logging, performance, debugging, Swarm, CI/CD, API	Operate, automate and troubleshoot

Ten-question self-check

Q 1. What is the difference between an image tag and a digest?

A A tag is a movable label; a digest is an immutable content hash.

Q 2. Why does the order of instructions in a Dockerfile matter?

A Layers are cached top to bottom; one changed instruction invalidates every later layer.

Q 3. What happens to data in a container's writable layer when it is removed?

A It is deleted with the container.

Q 4. Why can containers on the default bridge not resolve each other by name?

A Only user-defined networks provide Docker's embedded DNS.

Q 5. What does `-p 127.0.0.1:8080:80` do differently from `-p 8080:80`?

A It publishes the port only on the host's loopback interface.

Q 6. What is a multi-stage build for?

A Compiling in one stage and shipping only the result in a small final stage.

Q 7. What does `depends_on: condition: service_healthy` wait for?

A The dependency's health check to pass.

Q 8. Why use exec form for CMD?

A So your app is PID 1 and receives signals such as SIGTERM.

Q 9. What does exit code 137 indicate?

A SIGKILL, commonly the OOM killer or `docker kill`.

Q 10. What is the safest way to deliver a database password to a container?

A A mounted secret file or a secret manager, not an image or a plain environment variable.

Your lab environment

Everything here runs on a single machine. Use Linux (a VM or WSL 2 works well) for the networking, security and daemon topics, because Docker Desktop hides the real host behind a VM and some commands behave differently there.

Requirement	Recommended	Why
Docker Engine	29.x with the Compose and Buildx plugins	Matches these notes
Host	Ubuntu 24.04 or Debian 12/13 VM, 4 GB RAM, 30 GB disk	Real Linux kernel features visible
Tools	<code>curl</code> , <code>jq</code> , <code>git</code> , <code>dive</code> , <code>trivy</code> or <code>grype</code> , <code>cosign</code>	Used in later topics
Editor	VS Code with the Docker and Dev Containers extensions	Productivity
Registry account	GitHub (GHCR) or Docker Hub	Push and signing labs

```
BASH SNIPPET 8 lines
$ docker version --format '{{.Server.Version}} (API {{.Server.APIVersion}})'
29.8.2 (API 1.56)
$ docker compose version --short
5.5.1
$ docker buildx version
github.com/docker/buildx v0.37.1
$ docker info --format '{{.Driver}} | {{.CgroupVersion}} | rootless={{.SecurityOptions}}'
overlays | 2 | rootless=[name=seccomp,profile=builtin name=cgroupns]
```

Pro tip: Make a throwaway VM snapshot before the security and daemon labs. Several of them change firewall and daemon settings, and a snapshot lets you undo any mistake in seconds.

How the labs are organised

✓ Every topic ends with a lab or a set of review questions.

- ✓ Three **capstone projects** at the end combine many topics: a private registry with a signed pipeline, an observability stack and a hardened deployment.
- ✓ Commands that change the host (daemon settings, firewall, kernel parameters) are marked with a warning box.

Key takeaways

- Tier 2 assumes beginner fluency and adds production skills.
- Use a real Linux VM for the networking, security and daemon topics.
- Snapshot the VM before changing daemon or firewall settings.

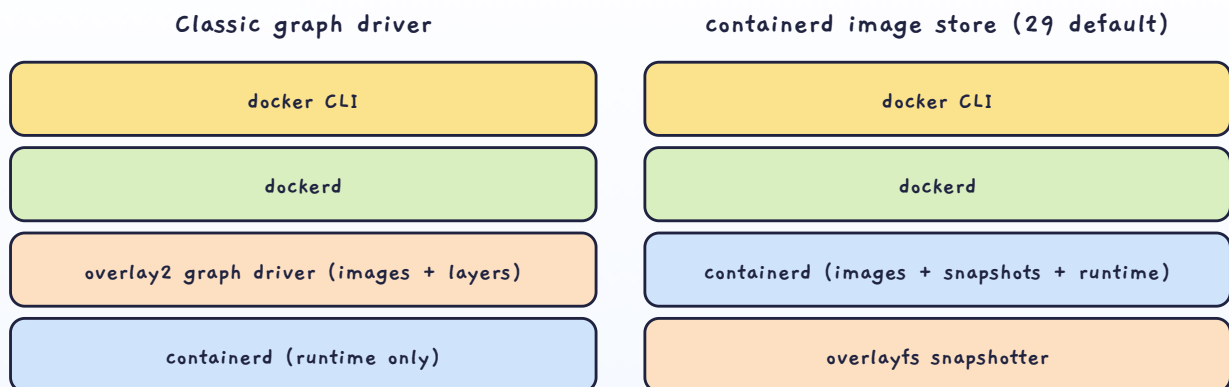
Docker Engine 29 in Practice

What the 29.x line changed (containerd image store, firewall backends, API floor), how to check your setup and how to plan an upgrade.

Docker Engine 29 is the current major line. Three changes affect how you work: the **containerd image store** is the default for new installs, an **experimental nftables firewall backend** exists, and the **minimum API version** was raised so very old clients no longer connect by default. Understanding them prevents surprises when your laptop, CI runner and servers run different versions.

DIAGRAM

Where image data lives: classic vs containerd image store



With the containerd store, one component owns images and containers, matching how Kubernetes nodes work.

The containerd image store

In the classic design **dockerd** stored layers itself using a *graph driver* (usually **overlay2**). With the containerd image store, **containerd** manages image content and snapshots; Docker talks to it. For you this brings:

- ✓ **Multi-platform images stored locally.** You can pull and keep several architectures of one tag and load OCI image indexes.
- ✓ **Attestations and refererrs** (SBOMs, provenance) are stored alongside images.
- ✓ **A path to lazy pulling** and peer-to-peer distribution through snapshotter plugins.
- ✓ **Alignment with Kubernetes**, where containerd already stores images.

BASH SNIPPET

5 lines

```
$ docker info --format '{{.DriverStatus}}'
[[driver-type io.containerd.snapshotter.v1]]
$ docker info | grep -iE 'storage driver|driver-type|snapshotter'
Storage Driver: overlayfs
driver-type: io.containerd.snapshotter.v1
```

Watch out: Existing installs keep their current store. Upgrading Docker does not migrate your images. If you switch an existing host to the containerd image store, images and containers pulled earlier are not visible in the new store until you re-pull or load them. Plan it as a migration: **docker save** what you need first.

```
JSON SNIPPET 3 lines

{
  "features": { "containerd-snapshotter": true }
}
```

Firewall backends: iptables and the new nftables option

Docker publishes ports and isolates networks by writing firewall rules. Historically that meant **iptables** (often via the iptables-nft compatibility layer). Engine 29 adds an **experimental native nftables backend**:

```
BASH SNIPPET 2 lines

$ sudo dockerd --firewall-backend=nftables      # or set "firewall-backend" in daemon.json
$ sudo nft list ruleset | grep -i docker
```

Backend	Status in 29	Use when
iptables (default)	Stable, widely tested	Production; Swarm; most hosts
nftables	Experimental	Modern distributions that are removing iptables; test labs

Experimental means rules, defaults and Swarm support may change between minor releases. Try it on a test host first and read the release notes before adopting it.

API version floor

Each Engine release speaks a range of **Engine API versions**. Engine 29 raised the minimum to **1.44** (the Docker 25 era), so clients older than Docker 25 can no longer connect unless the daemon is explicitly configured to allow older versions. Tools that pin ancient API versions (old CI plugins, monitoring agents, **docker-py** 4.x) are the usual casualties.

```
BASH SNIPPET 5 lines

$ docker version --format 'client API {{.Client.APIVersion}} / server API {{.Server.APIVersion}} (min {{.Server.MinAPIVersion}})'
client API 1.56 / server API 1.56 (min 1.44)

$ DOCKER_API_VERSION=1.40 docker ps
Error response from daemon: client version 1.40 is too old. Minimum supported API version is 1.44
```

Checking and planning an upgrade

Step	Command / action
Read the release notes for every version you skip	docs.docker.com/engine/release-notes/29/ and the deprecated features page
Inventory clients and tools that talk to the socket	<code>ss -xp grep docker.sock</code> , CI configs, monitoring agents
Back up volumes and save critical images	<code>docker save</code> , volume tar backups (Beginner notes, Topic 32)
Upgrade a canary host first	Run representative workloads for a few days
Use live-restore to keep containers running during daemon restarts	<code>"live-restore": true</code> in <code>daemon.json</code>
Verify after the upgrade	<code>docker info</code> , <code>docker ps</code> , health checks, firewall rules