

Docker Advanced

Visual handwritten study guide - Tier 3

Understand

Clear, compact explanations

Visualize

Diagrams that show the moving parts

Practice

Readable commands and code

Remember

Highlighted revision points

60 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 - Container Internals

1	Welcome to Docker Advanced	2	What a Container Really Is
3	Namespaces in Depth	4	cgroups v2 Internals
5	Build a Container by Hand	6	The OCI Runtime Spec and runc
7	The OCI Image and Distribution Specs		

Part 2 - The Runtime Stack

8	containerd Architecture	9	dockerd Internals and /var/lib/docker
10	CRI, Kubernetes and the Daemonless Alternatives	11	Rootless Internals: How It Works
12	Alternative Runtimes in Depth	13	GPUs, Devices and WebAssembly
14	Docker Plugins: Volume, Network, Logging and AuthZ		

Part 3 - BuildKit and Image Engineering

15	LLB and the BuildKit Solver	16	Frontends, Named Contexts and Build Orchestration
17	Build Farms: Remote and Scalable BuildKit	18	Cache Keys and Debugging Invalidation
19	OCI Artifacts, Referrers and ORAS	20	Lazy Pulling, Compression and Distribution at Scale
21	Reproducible Builds	22	Base Image Governance at Scale

Part 4 - Advanced Networking

23	netfilter: What Docker Writes to the Firewall	24	Connection Tracking, NAT and Port Exhaustion
25	Traffic Control, Bandwidth and Chaos	26	CNI Plugins Outside Kubernetes
27	eBPF Networking and Observability	28	Multi-Host Networking Without Kubernetes
29	High-Availability Entry Points		

Part 5 - Storage and State

30	Union Filesystems: overlay2 Internals	31	Volume Drivers and Network Storage
32	Databases in Containers: Doing It Properly	33	Backup, Snapshots and Disaster Recovery
34	Disk, I/O and Inode Management	35	Stateful Workload Patterns

Part 6 - Advanced Security

36	Threat Modelling a Container Platform	37	Writing Custom seccomp Profiles
38	AppArmor and SELinux Profiles	39	User Namespaces and Privilege Reduction
40	Container Escapes: Case Studies and Defences	41	Signing, Attestations and Provenance
42	Secrets Management at Scale	43	Runtime Detection, Auditing and Forensics
44	Benchmarks, Compliance and a Hardening Baseline		

Part 7 - Platform and Scale

45	Capacity Planning and Density	46	Choosing an Orchestrator: Compose, Swarm, Nomad or Kubernetes
47	Observability Stack for Container Hosts	48	Performance Engineering Inside Containers
49	Docker in CI: Runners, Caching and Isolation	50	An Internal Platform: Golden Paths and Standards
51	Cost, Efficiency and Image Economics	52	Migrations: Legacy Apps, Engine Upgrades and Moving to Kubernetes

Part 8 - Capstone Projects and Reference

PROJECT 1	Project 1: A Hardened Docker Host, Provable by Script	PROJECT 2	Project 2: Two Hosts, One Service, Survive a Failure
PROJECT 3	Project 3: A Supply Chain Gate That Actually Blocks	PROJECT 4	Project 4: A Game Day
57	Tier 3 Review Questions	APPENDIX A	Command and Tool Reference for Docker Operators
APPENDIX B	Troubleshooting Playbooks	APPENDIX C	Glossary and Where to Go Next

Welcome to Docker Advanced

What this tier assumes, what it teaches, and the mindset: containers are Linux features you can inspect, rebuild and break.

Tier 3 is for people who already run containers in production and want to understand them to the bottom: the kernel features underneath, the runtime stack between `docker run` and the process, the image and registry formats, advanced build infrastructure, deep security, and the design of container platforms for whole organisations.

This tier assumes everything in the Beginner and Intermediate notes. If terms like *digest*, *cache mount*, *user-defined network*, *cap-drop* or *health check* are not automatic for you, go back first. Advanced notes move fast and skip the basics.

Part	Focus	After it you can...
1. Container internals	Namespaces, cgroups, OCI runtime and image specs	Build a container by hand and explain every kernel object involved
2. The runtime stack	containerd, dockerd internals, CRI, rootless, alternative runtimes, plugins	Know which component does what, and swap them
3. BuildKit and image engineering	LLB, build farms, cache keys, OCI artifacts, lazy pulling, reproducibility	Design build infrastructure
4. Advanced networking	netfilter, conntrack, tc, CNI, eBPF, HA ingress	Debug packets at the kernel level
5. Storage and state	overlayfs internals, quotas, replication, checkpoint/restore	Reason about data durability and I/O
6. Advanced security	Escape classes, custom profiles, userns, LSMs, eBPF security, supply-chain attacks	Threat-model and harden a platform
7. Platform and scale	Golden paths, build farms, HA, migration to Kubernetes, kernel-level debugging	Run containers as a product for other teams
8. Projects and reference	Write a mini runtime, a build farm, an incident exercise	Prove it with working systems

How to use these notes

- ✓ Do the labs on a **disposable Linux VM** you can snapshot. Many commands need root and change kernel state (namespaces, cgroups, firewall rules).
- ✓ Kernel version matters. Examples assume a recent LTS kernel (6.x) with **cgroup v2**. Check yours with `uname -r` and `stat -fc %T /sys/fs/cgroup` (`cgroup2fs` means v2).
- ✓ Where a feature is **experimental** or changes between releases, the notes say so. Verify against current documentation before relying on it in production.
- ✓ Nothing here replaces reading the specifications: the OCI runtime, image and distribution specs, and the Linux man pages (`namespaces(7)`, `cgroups(7)`, `capabilities(7)`) are short and authoritative.

Prerequisite self-check

Q What does BuildKit do with a stage nothing depends on?

A It skips it entirely.

Q Why is a digest different from a tag?

A A digest identifies exact content; a tag is a movable pointer.

Q What does `--cap-drop ALL` remove?

A All default Linux capabilities from the container's process.

Q Why does `-p 8080:80` bypass ufw?

A Docker's DNAT and FORWARD rules apply before the INPUT chain that ufw manages.

Q What makes exit code 137?

A SIGKILL, commonly the OOM killer.

Q What is the writable layer for?

A Small temporary changes; real data belongs on volumes.

Lab environment

Item	Recommendation
OS	Debian 12/13 or Ubuntu 24.04 VM, 4 vCPU, 8 GB RAM, 40 GB disk, nested virtualisation if you want Kata
Docker	Engine 29.x with Buildx and Compose
Tools	<code>strace</code> , <code>perf</code> , <code>bpfftrace</code> , <code>iproute2</code> , <code>nftables</code> , <code>conntrack</code> , <code>tcpdump</code> , <code>jq</code> , <code>skopeo</code> , <code>crane</code> , <code>oras</code> , <code>runc</code> , <code>containerd</code> with <code>nerdctl</code> , <code>buildctl</code> , <code>dive</code> , <code>cosign</code> , <code>trivy</code>
Kernel features	cgroup v2, user namespaces enabled, eBPF (BTF) for the security and tracing labs

BASH SNIPPET

6 lines

```
$ uname -r && stat -fc %T /sys/fs/cgroup
6.8.0-48-generic
cgroup2fs
$ sysctl kernel.unprivileged_userns_clone user.max_user_namespaces 2>/dev/null
user.max_user_namespaces = 63704
$ docker version --format '{{.Server.Version}}' && runc --version | head -1 && containerd --version
```

Watch out: Do the kernel and firewall labs on a VM, never on your daily-driver laptop or a shared server. A wrong `nft flush ruleset` or a runaway cgroup can lock you out of the machine.

Key takeaways

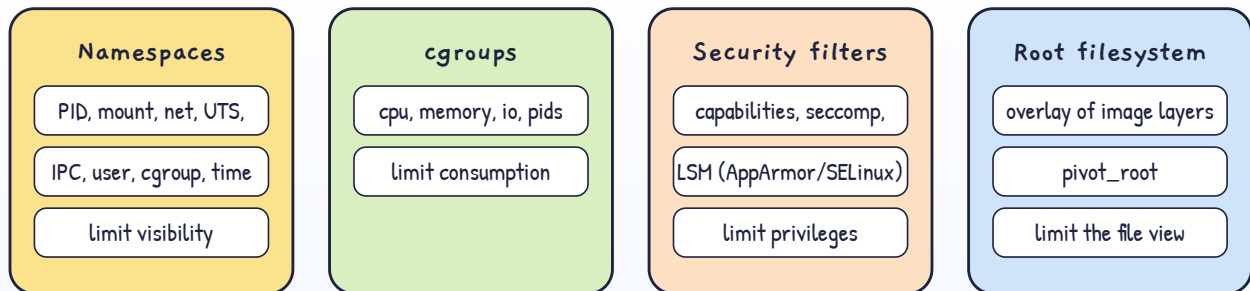
- Tier 3 assumes production fluency and goes to the kernel, runtimes, specs and platform design.
- Use a snapshot-able Linux VM with cgroup v2 and a recent kernel.
- Prefer specifications and man pages as ground truth.

What a Container Really Is

A container is a process with a restricted view and restricted resources. Prove it from the host.

There is no kernel object called 'container'. A container is an ordinary process (and its children) that has been placed into a set of **namespaces** (what it can *see*), a **cgroup** (what it can *use*), and wrapped with **security filters** (what it may *do*), and whose root filesystem was swapped for an image's contents.

DIAGRAM The four ingredients



Docker, containerd and runc are tools that assemble these four for you.

Proof: the container's process is visible on the host

```
BASH SNIPPET 13 lines

$ docker run -d --name demo --memory 128m alpine sleep 1000
$ PID=$(docker inspect -f '{{.State.Pid}}' demo); echo $PID
20511
$ ps -o pid,ppid,user,cmd -p $PID
  PID  PPID  USER   CMD
20511  20488  root    sleep 1000
$ ps -o pid,cmd -p $(ps -o ppid= -p $PID)      # its parent: the containerd shim
  PID  CMD
20488  /usr/bin/containerd-shim-runc-v2 -namespace moby -id 6f2c...

$ docker exec demo ps
  PID  USER   COMMAND
    1  root    sleep 1000      # PID 1 inside, PID 20511 outside
```

Its namespaces

```
BASH SNIPPET 11 lines

$ sudo ls -l /proc/$PID/ns
lrwxrwxrwx 1 root root 0 cgroup -> 'cgroup:[4026532598]'
lrwxrwxrwx 1 root root 0 ipc    -> 'ipc:[4026532596]'
lrwxrwxrwx 1 root root 0 mnt    -> 'mnt:[4026532594]'
lrwxrwxrwx 1 root root 0 net    -> 'net:[4026532599]'
lrwxrwxrwx 1 root root 0 pid    -> 'pid:[4026532597]'
lrwxrwxrwx 1 root root 0 time   -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 user   -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 uts    -> 'uts:[4026532595]'

$ sudo ls -l /proc/1/ns | grep -E 'user|time'      # host: same numbers = shared
```

Each symlink names a namespace by inode number. Two processes with the same number for a type share that namespace. Here the container has its own `mnt`, `net`, `pid`, `ipc`, `uts` and `cgroup` namespaces but shares the host's `user` and `time` namespaces (default Docker, no `users-remap`).

Its cgroup, capabilities and filters

```
BASH SNIPPET 14 lines

$ cat /proc/$PID/cgroup
0::/system.slice/docker-6f2c...scope
$ cat /sys/fs/cgroup/system.slice/docker-6f2c...scope/memory.max
134217728
$ grep -E 'Cap(Prm|Eff|Bnd)|Seccomp|NoNewPrivs' /proc/$PID/status
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
CapBnd: 00000000a80425fb
NoNewPrivs: 0
Seccomp: 2
$ capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_s
etpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
$ cat /proc/$PID/attr/current
docker-default (enforce) # AppArmor profile, if enabled
```

Its root filesystem

```
BASH SNIPPET 3 lines

$ sudo ls -l /proc/$PID/root/ | head -4 # the container's / as the host sees it
$ sudo grep -m3 overlay /proc/$PID/mountinfo
$ sudo cat /proc/$PID/mountinfo | grep ' / / ' | cut -c1-200
```

Property	Kernel interface	Where to look
Process identity	task struct, PID namespace	<code>/proc/<pid>/status</code> (NSpid: shows PIDs per level)
Visibility	namespaces	<code>/proc/<pid>/ns/*</code> , <code>lsns</code>
Resources	cgroup v2	<code>/proc/<pid>/cgroup</code> , <code>/sys/fs/cgroup/...</code>
Privileges	capabilities, seccomp, LSM, no_new_privs	<code>/proc/<pid>/status</code> , <code>attr/current</code>
Filesystem	mount namespace + overlayfs	<code>/proc/<pid>/mountinfo</code> , <code>root</code>
Identity mapping	user namespace	<code>/proc/<pid>/uid_map</code> , <code>gid_map</code>

```
BASH SNIPPET 8 lines

$ grep NSpid /proc/$PID/status
NSpid: 20511 1 # PID 20511 on the host is PID 1 in the container
$ lsns -p $PID
NS TYPE NPROCS PID USER COMMAND
4026531837 user 143 1 root /sbin/init
4026532594 mnt 1 20511 root sleep 1000
4026532595 uts 1 20511 root sleep 1000
...
```

This is why 'container escape' is a kernel-boundary problem, and why a kernel vulnerability affects every container on the host. It is also why you can debug containers with ordinary host tools: `strace -p`, `perf`, `nsenter`, `gdb`.

1. Start a container with memory and CPU limits and a read-only root.
2. From the host, find its PID, namespaces, cgroup path, capability masks and seccomp mode.
3. Compare the container's `/proc/<pid>/ns` links with PID 1's and list which namespaces are shared.
4. Use `nsenter -t <pid> -m -u -i -n -p sh` to enter the container with host tools, and explain what each flag selects.

Key takeaways

- A container is a process with namespaces, a cgroup, security filters and a swapped root.
- Everything is inspectable from the host via `/proc` and `/sys`.
- Shared kernel means kernel bugs cross container boundaries.