

Containers

Aayush Bajaj

July 10, 2026

Contents

1	what a container actually is	1
1.1	namespaces: the six that matter	1
1.2	cgroups v2: the accountant	2
1.3	not a vm	2
2	images and layers	2
2.1	union filesystems and copy-on-write	2
2.2	content-addressable everything	3
3	dockerfile mechanics	3
3.1	the cache rules	3
3.2	multi-stage builds, worked	3
4	networking	4
5	volumes	5
6	registries and digests	5
7	security, one paragraph	5
8	orchestration, one honest paragraph	5
9	the oci spec	6
10	see also	6

a container is not a small virtual machine. it is an ordinary linux process (tree) that the kernel has been told to *lie to* — about what processes exist, what the filesystem looks like, what the network is, who root is — plus an accountant capping what it may consume. the lying is **namespaces**, the accounting is **cgroups**, and everything else (images, registries, orchestrators) is packaging around those two syscall families (Tanenbaum, Andrew S., 2008).

1 what a container actually is

1.1 namespaces: the six that matter

a namespace wraps one global kernel resource so a process group gets a private view of it. spawn a process with `clone()` or `unshare()` flags and it lives in new namespaces:

namespace	isolates	inside the container you see
pid	process ids	your init is pid 1; other processes don't exist
mnt	mount table	a private filesystem tree (the image)
net	network stack	own interfaces, routes, ports, firewall
uts	host + domain name	your own hostname
ipc	sysv ipc, posix queues	private shared-memory segments
user	uid/gid mappings	"root" inside maps to a nobody outside

(two more exist — **cgroup** and **time** — worth knowing but rarely the point.) the **user** namespace is the security-critical one: it lets a container's uid 0 be an unprivileged uid on the host, which is what "rootless" means below.

1.2 cgroups v2: the accountant

- **control groups** meter and cap resource use per process-tree: **cpu.max** (bandwidth quota), **memory.max** (hard cap — exceed it and the cgroup's own oom killer fires), **io.max**, **pids.max** (fork-bomb fence).
- v2's improvement is a single unified hierarchy: one tree under **/sys/fs/cgroup**, every controller attached to the same nodes, so "this pod gets 2 cpus and 4 gb" is one subtree — the primitive kubernetes resource limits compile down to.
- namespaces decide what you can *see*; cgroups decide what you can *spend*. isolation needs both — a container that can see nothing but eat all the ram is still a noisy neighbour.

1.3 not a vm

dimension	container	virtual machine
kernel	shared with host	own guest kernel
boots in	milliseconds (it's a fork)	seconds-plus (it's a machine)
size	mb (layers, shared)	gb (disk image)
density	hundreds per host	tens per host
isolation	one kernel = one attack surface	hypervisor boundary, much smaller
os mixing	linux-on-linux only	anything on anything

the isolation row is the honest one: a kernel exploit escapes every container on the host, which is why multi-tenant clouds run containers *inside* micro-vms (firecracker, kata) — taking both columns.

docker on macos — including the demos below — quietly runs a linux vm and containers inside it; there are no linux namespaces on darwin

2 images and layers

2.1 union filesystems and copy-on-write

- an image is a stack of read-only **layers**, each a tarball of filesystem changes. at run time, **overlayfs** merges them: the layers become read-only **lowerdir**s, a fresh empty **upperdir** goes on top, and the container sees the union.
- writes are **copy-on-write**: modifying a file from a lower layer first copies it up to **upperdir**, then edits the copy. deletes write a **whiteout** entry that masks the lower file. the image is never touched — which is why 200 containers can share one image at the cost of one, and why anything written to the container layer dies with the container (volumes below are the escape hatch).

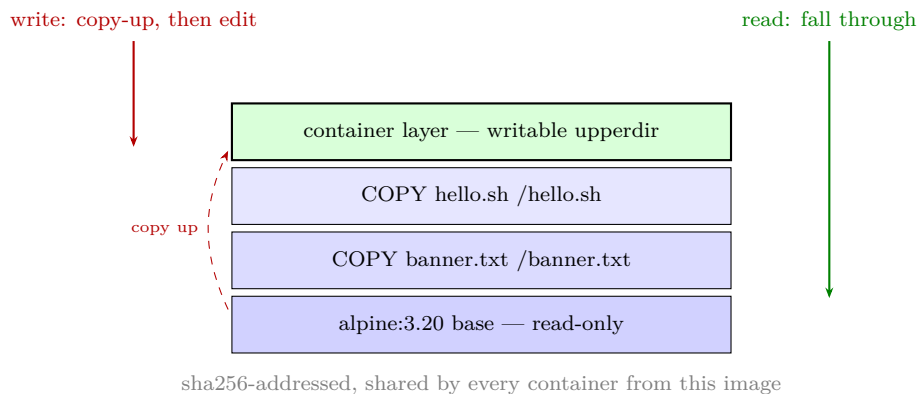


Figure 1: overlayfs at runtime: read-only image layers below, one writable layer on top. reads fall through to the highest layer holding the file; writes copy up first.

2.2 content-addressable everything

layers and manifests are named by the sha256 of their bytes. the demo image built below really is just three digests:

```
$ docker image inspect layerdemo --format '{{range .RootFS.Layers}}{{println .}}{{end}}'
sha256:88b4fba61c4c714a2fc173ddf7e9324a257304696830bf41ad28ecc58c11c95f
sha256:e48e9d6daab227918910cc87acc76d6ca91c822d44ba4633bbe0354ac5531381
sha256:6e1631f50f3a107cc640c8591071adc8f4df45704b5181756ac7ae78d9028b41
```

identical layers dedupe on disk, in transfer, and in the registry — the same content-addressing bet as git and the [build-system caches](#).

3 dockerfile mechanics

3.1 the cache rules

each instruction produces one layer, and the builder replays instructions top-down against its cache: an instruction is a cache hit if the instruction text is unchanged *and* (for COPY and ADD) the checksums of the copied files are unchanged. **the first miss invalidates everything after it** — layers below depend on layers above, exactly the dirty-cone propagation from [build systems](#).

the practical layering discipline follows immediately: order layers by change frequency — base image, then system packages, then dependency manifests (`requirements.txt`, `package.json`) + install, then your source code *last*, so the daily edit invalidates one cheap COPY instead of re-running `pip install`.

3.2 multi-stage builds, worked

ship the artefact, not the toolchain: build in one stage, COPY --from the result into a minimal runtime stage. executed:

```
1 # ---- stage 1: "build" ----
2 FROM alpine:3.20 AS build
3 WORKDIR /src
4 RUN echo "banner built in stage 1" > banner.txt # heavy toolchain lives here
5
6 # ---- stage 2: runtime (ships without the toolchain) ----
7 FROM alpine:3.20
8 COPY --from=build /src/banner.txt /banner.txt
```

```

9 COPY hello.sh /hello.sh
10 CMD ["sh", "/hello.sh"]

```

```

== build 2 (nothing changed):
#6 CACHED
#7 CACHED
#8 CACHED
== build 3 (hello.sh edited):
#6 [build 3/3] RUN echo "banner built in stage 1" > banner.txt
#6 CACHED
#8 [stage-1 2/3] COPY --from=build /src/banner.txt /banner.txt
#8 CACHED
#9 [stage-1 3/3] COPY hello.sh /hello.sh
== run it:
banner built in stage 1
hello from pid 1 on b8c22872ae9a
one more line

```

read the trace: the unchanged build stage and banner **COPY** stayed cached; editing **hello.sh** re-ran only its own **COPY** layer. and the run line quietly demonstrates two namespaces — the shell is **pid 1** (pid namespace) on a random hostname (uts namespace). more directly:

```

$ docker run --rm alpine:3.20 sh -c 'ps; hostname'
PID   USER      TIME  COMMAND
    1   root         0:00 sh -c ps; hostname
    7   root         0:00 ps
d5493fe8cb30

```

a machine running hundreds of processes, and inside the namespace: two.

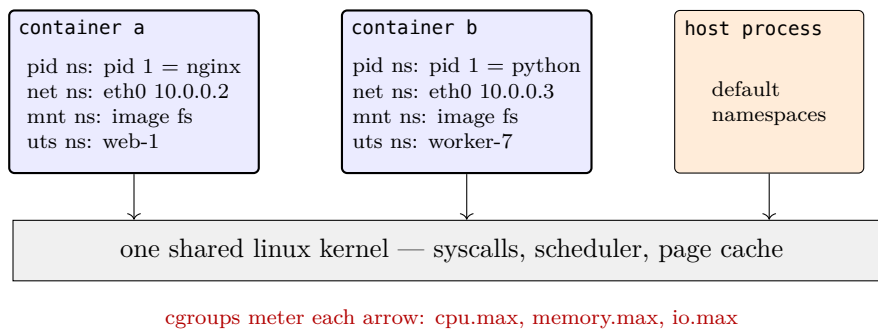


Figure 2: one kernel, many worlds: each container’s namespaces wrap its view of pids, mounts, network and hostname; cgroups meter each box. nothing is virtualised — every syscall lands in the same kernel.

4 networking

- **bridge** (default): each container gets a network namespace with a **veth pair** — one end inside as **eth0**, the other plugged into a software bridge on the host; outbound traffic is nat’d, inbound needs explicit publishing (**-p 8080:80** programs a dn timerule). containers on the same bridge reach each other by ip (and by name, on user-defined bridges).
- **host**: no network namespace at all — the container shares the host stack, ports

and all. fast, and exactly as isolated as it sounds.

- **none**: a loopback and silence; you build the rest yourself.
- **container:x** / pod networking: join another container's network namespace — how kubernetes pods give all their containers one ip and a shared **localhost**. see [internet networks](#) for the stack underneath.

5 volumes

- the writable layer is a scratchpad that dies with the container; anything worth keeping lives in a **volume** — a host-managed directory mounted into the container's mnt namespace, bypassing overlayfs (and its copy-up cost) entirely.
- **bind mounts** map an explicit host path (great for development: source code mounted live); **named volumes** let the engine own the path (right for databases); **tmpfs** mounts keep secrets in ram only.
- rule of thumb: image = code, immutable; volume = state, precious; container layer = cache, disposable.

6 registries and digests

- a registry is an http content store for image manifests and layer blobs; **push** and **pull** move only the layers the other side lacks (content addressing again).
- **tags are mutable pointers**: `python:3.12` can point at different bytes tomorrow. **digests are immutable**: `alpine@sha256:d9e853e87e5552...` (the real digest of the image the demos above pulled) names exactly one manifest forever.
- production discipline: humans read tags, machines pin digests — ci resolves the tag once and deploys by digest, or the supply chain inherits a mutable dependency. the same hermeticity argument as [reproducible builds](#), one level up.

7 security, one paragraph

defence is subtraction. **rootless**: run the engine and containers in a user namespace so container-root is a nobody on the host — a breakout lands unprivileged. **capabilities**: root's powers are roughly forty separate flags (`CAP_NET_ADMIN`, `CAP_SYS_ADMIN`, ...); runtimes drop most by default and you should drop the rest (`--cap-drop=ALL` plus the two you need). **seccomp**: a syscall allowlist filter — the default profiles block a few dozen exotic syscalls, closing most historical kernel-escape routes. add read-only root filesystems and no-new-privileges, and the honest residual risk is the shared kernel itself — which is the vm row in the earlier table, and why the paranoid run micro-vms.

8 orchestration, one honest paragraph

one host runs containers; a fleet needs a scheduler. kubernetes' model in one breath: the unit is the **pod** (containers sharing network and volumes, scheduled together); you declare desired state (**deployment: replicas: 3**) and **controllers** run reconciliation loops that diff desired against actual and act — a crashed pod is rescheduled not because something noticed the crash, but because the loop noticed the count

was 2. services give pods a stable virtual ip (the [service discovery](#) problem, solved by the platform). that is genuinely the whole idea; the other 900 pages are api surface.

9 the oci spec

“docker” stopped being one thing in 2015: the **open container initiative** standardises the **image format** (layers + manifest + config), the **runtime spec** (what a bundle is and what **create**, **start** and **kill** mean), and the **distribution spec** (the registry http api). consequence: images build with docker or buildah, run on containerd or cri-o or podman, and any registry serves them — the formats are the interface, the tools compete. podman’s whole pitch is oci compatibility without a root daemon.

10 see also

- [microservices](#) — what all these boxes are deployed to run
- [build systems](#) — layer caching is the build dag’s staleness problem in a trenchcoat
- [linux](#) — the kernel these processes all share
- [concurrency](#) — isolation by process when threads share too much

references

Tanenbaum, Andrew S. (2008). *Modern Operating Systems*, Pearson.