Containers 1/3: DevOps and Why Containers

From waterfall to DevOps; containers vs virtual machines

Walter Rudametkin

Walter.Rudametkin@univ-rennes.fr

https://rudametw.github.io/teaching

ESIR 2 SI — Architectures Logicielles

Université de Rennes













Lecture 1: Road Map

1. DevOps and Why Containers
today · lab Module 0

2. Containers in Practice
lab Modules 2–4

3. Under the Hood and Production
lab Modules 0–1 and 5

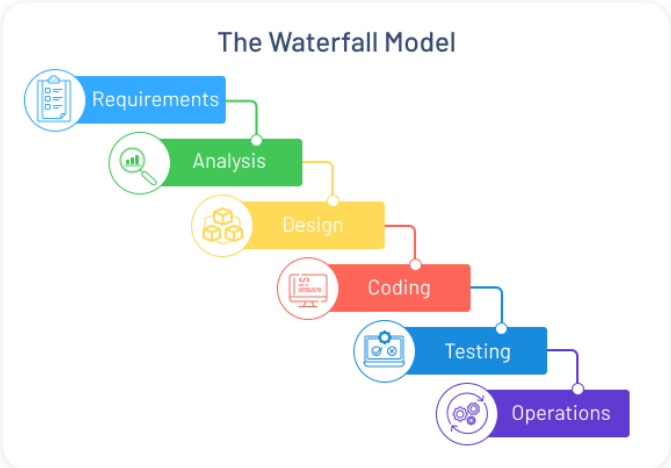
1. **DevOps** — from waterfall, CI/CD, tools
2. **Microservices** — cloud native, scaling
3. **Why Containers** — dependencies, shipping analogy
4. **Containers vs VMs** — shared kernel, trade-offs
5. **Image vs Container** — layers, vocabulary

lab Module 0

Goal: why the industry moved to containers, and what a container really is.

appendix: deep dives **field guide**: recipes, troubleshooting

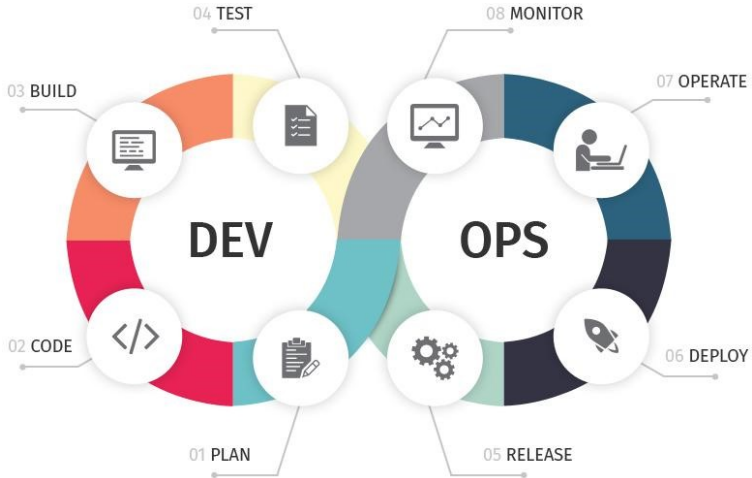
The Waterfall Model



Slow hand-offs: dev writes it, ops must make it run… months later.

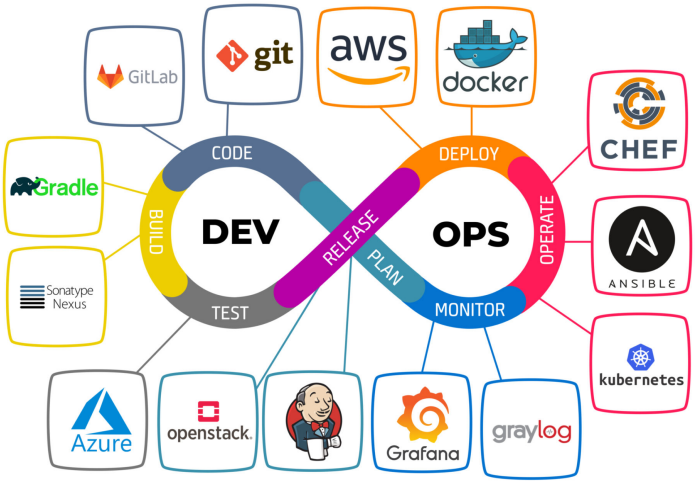

WORKED FINE IN DEV
OPS PROBLEM NOW
quickmeme.com

DevOps = Development + Operations



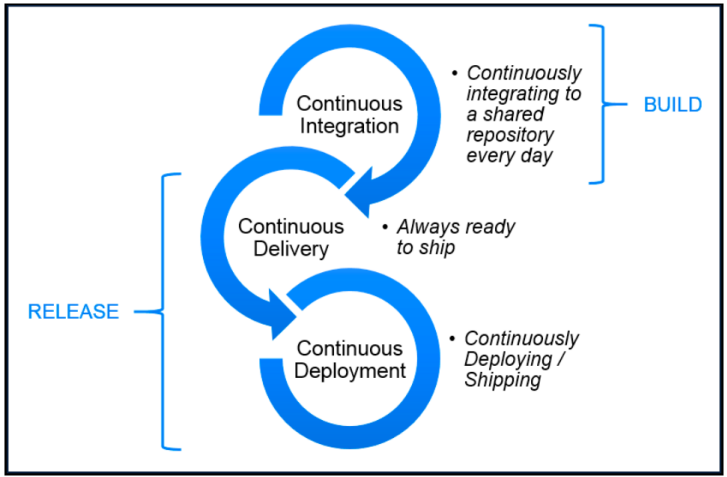
One team owns the whole loop, from plan and code to deploy and monitor.

Popular Tools



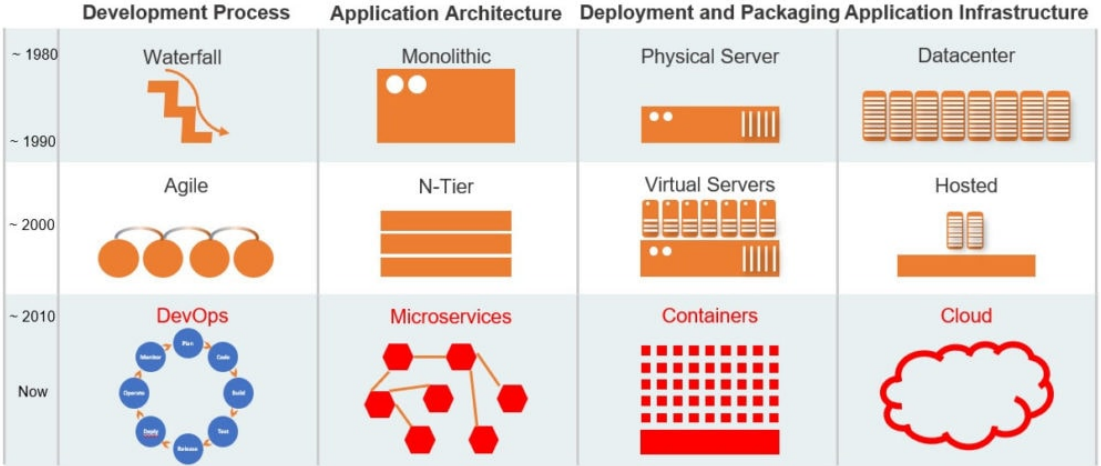
Every stage is automated; containers carry the app from build to run.

Continuous Integration, Delivery, Deployment



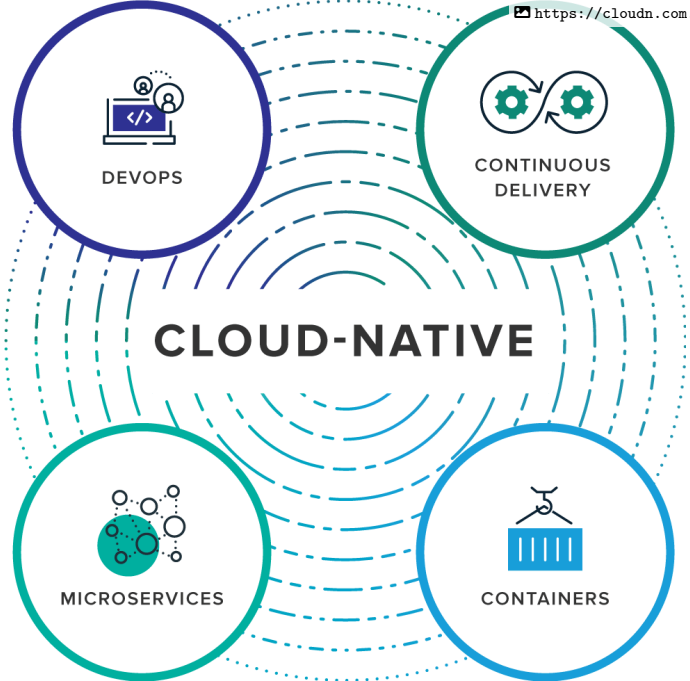
Integrate daily, always ready to ship: the same environment at every step.

Software Architecture Over Time



Monolith on physical servers → microservices in containers, in the cloud.

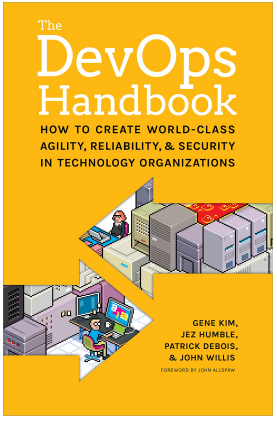
https://cloudn.com.au/cloud-native-approach.html



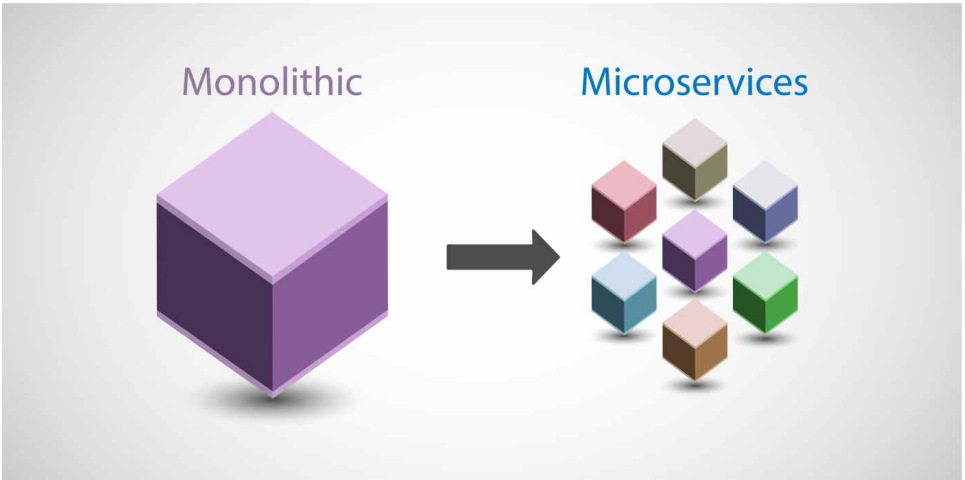
Reading

DevOps: A Software Architect's Perspective
by Ingo Weber, Len Bass & Liming Zhu

The DevOps Handbook
by Gene Kim, Jez Humble, Patrick Debois, John Willis & John Allspaw

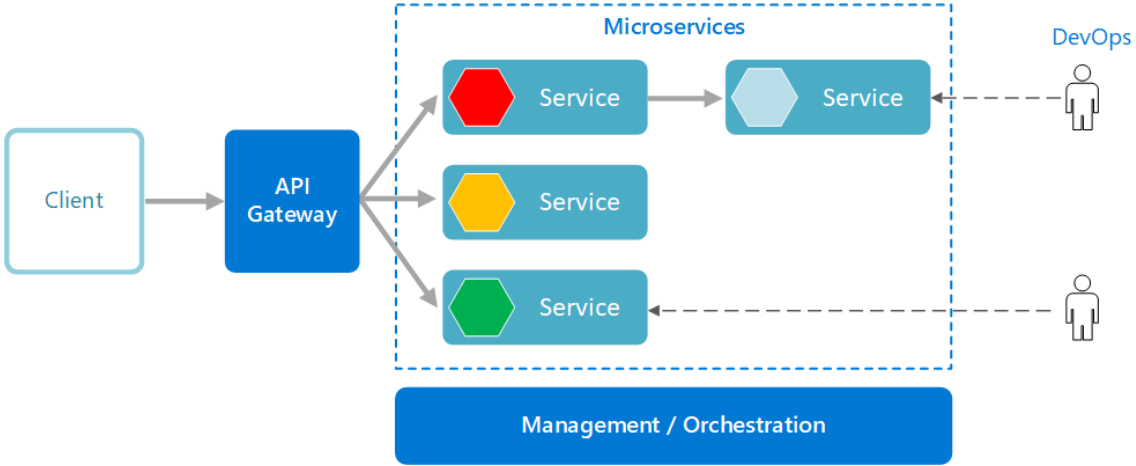


Towards Microservices



Split the monolith into small services, each deployed and scaled on its own.

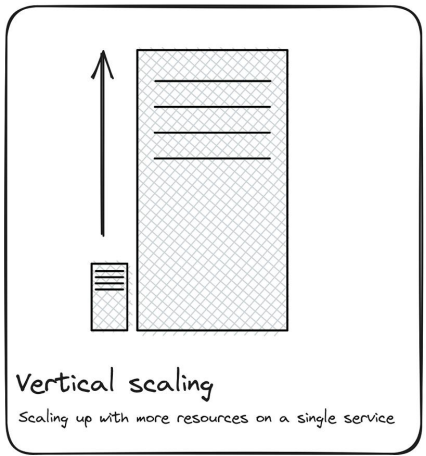
Microservice Architecture

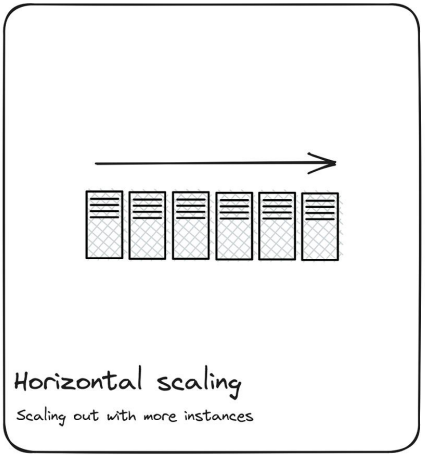


Dozens of services × environments × daily deploys…

… we need one standard, self-contained unit: the **container**

Scaling: Vertical vs Horizontal





scalablebackend.com

Containers make horizontal scaling cheap: start more identical instances.

Docker, Podman & Friends





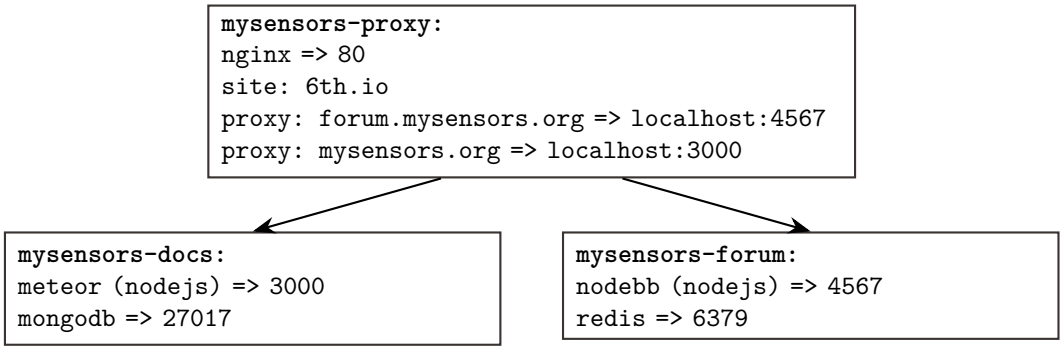
podman

- **2013**: **Docker** (dotCloud) makes Linux containers easy: **build**, **ship**, **run**
  - Docker Engine: open source (Moby, Apache-2.0)
  - Docker Desktop: proprietary, paid above 250 employees or $10M revenue
- **2015**: the **OCI** (Open Container Initiative) standardizes images and runtimes
- **2019**: Podman 1.0 (Red Hat): daemonless, rootless, same CLI (lab)
- Built on Linux kernel features: namespaces, cgroups, layered filesystems
- Ancestors: `chroot` (1979), FreeBSD Jails (2000), Solaris Zones (2004), LXC (2008) (rkt: a competitor, archived 2020)

Why Containers?

Problem

- An application has many dependencies that need to coexist
- Deployments with different demands: development/CI, test, production



Lecture 2 describes exactly this kind of stack in one `compose.yaml`.

Application Deployment Got Complex

Application stack

- basic OS
- JVM
- static web server
- front-end platform
- database layer
- application code



Deployment environments

- development VM
- QA server
- customer data center
- public cloud
- contributor's laptop
- production servers
- production clusters

Every combination must work…

installing, versioning, configuring, testing: on every machine, again and again

Analogy: Shipping Goods Before and After 1960

Before 1960



every kind of goods $\times$ every transport

Solution: the shipping container



- **Separation of concerns**
  - the user cares about packing the inside
  - the shipper cares about moving the container
- **Standardized interface**

Containers: A Standardized Interface



Standardized interface for software containers

Developer concerns (inside)

- code, libraries, services
- configuration, data
- *all servers look the same*

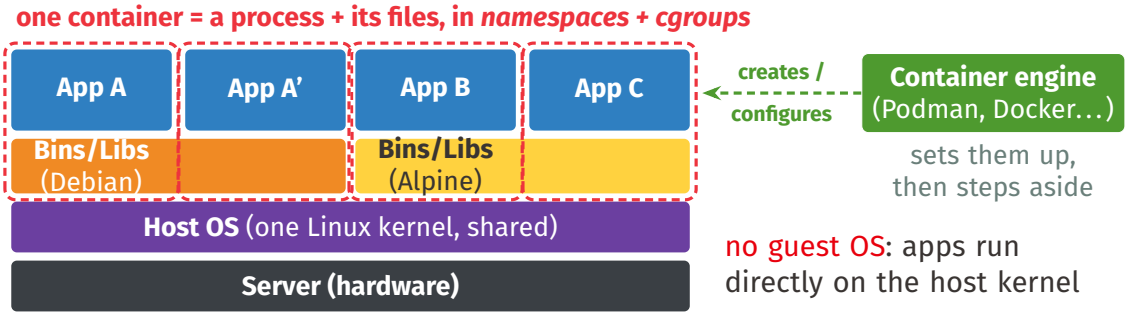
Ops concerns (outside)

- moving, starting/stopping containers
- logging, monitoring, network config.
- *all containers look the same*

N apps $\times$ M environments $\Rightarrow$ $N + M$

package each app once, teach each environment to run containers once

What Is a Container?



Containers are an **abstraction at the application layer** that packages code and dependencies together.

Images built `FROM` the same base share those read-only layers (on disk and in RAM).

Containers in 2026: The Default Unit of Deployment

Why the hype? ship software *with* its environment; one command runs a service; just processes (no guest kernel)

Where they run

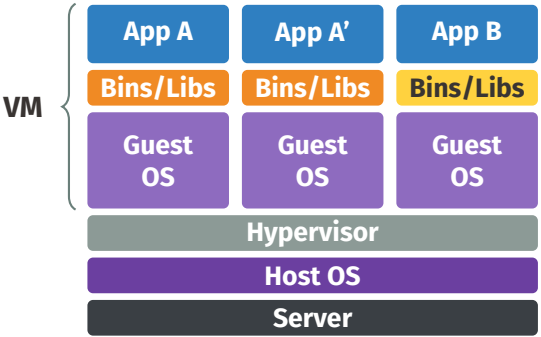
- Kubernetes, on any cloud or on-premises
- "serverless" (Cloud Run, Fargate)
- CI jobs, a fresh container each
- your own VM + Podman (assignment!)

What you use them for

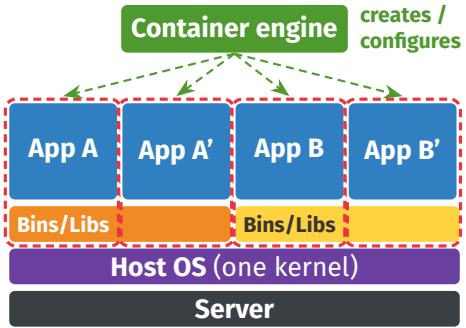
- any service version, one command
- test on several distros, no VMs
- reproducible dev environments, CI
- integration tests (Testcontainers)

The image is what you build, test, ship and deploy.

Containers vs Virtual Machines



Virtual machines
each VM boots its own guest kernel



Containers
share the host kernel (+ bins/libs)

Hypervisor types (bare metal vs hosted): appendix.

Containers vs VMs: Trade-offs

Containers

- share the host OS kernel
- isolate the app's processes
- less disk and RAM: cheaper
- fast to start (no boot)
- typically a single app

Virtual machines

- different OSes side by side
- stronger isolation
- stricter resource controls
- multiple apps

Both package an app ("works on one machine, works on all") and scale it.

Not either/or: in the cloud and on Windows/macOS, containers run *inside* VMs. VM-grade isolation, container UX: **Kata Containers**, **Firecracker** (micro-VMs), **gVisor** (user-space kernel).

Linux Containers Need a Linux Kernel

- Linux images hold Linux binaries (ELF, Linux system calls)
- **Linux laptop**: native
- **Windows**: WSL2, a real Linux kernel in a light Hyper-V VM
- **macOS**: `podman machine` boots a Fedora CoreOS VM
  (Docker Desktop also hides a Linux VM)
- **Not like a JVM** (portable bytecode, any OS):
  an image is tied to a kernel (and a CPU architecture)

WSL2: work in `~`, never in `/mnt/c`
(slow 9P network filesystem, no xattrs for OverlayFS)

Lab: Module 0

Feel Like VMs, but Closer to `chroot` on Steroids

VM-like

- get a shell
- own process space
- own network interface
- root *inside*
  (mapped to your UID when rootless)
- install packages
- run services
- change routing, firewall…

Unlike a VM

- host kernel (same OS!)
  - shared kernel modules
- doesn't need `init`
- doesn't need `syslogd`, `cron`…

It's just normal processes on the host machine

contrast with "opaque" VMs (lecture 3 proves it: `ps` on the host, `uname -r`)

Image vs Container

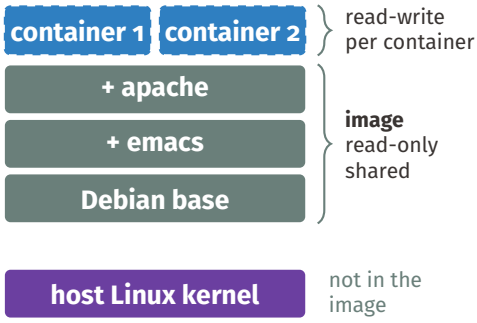


Image	Container
program on disk	→ process
class	→ object (instance)
read-only layers	→ + one writable layer
+ metadata (cmd, ports…)	
built once, shipped	→ run many times

```
>_ podman images    lists images
>_ podman ps -a     lists containers
```

Each build instruction (`RUN`, `COPY`…) adds a layer (lecture 2); the layers are stacked by OverlayFS (lecture 3).

Lecture 1: What to Remember

- **DevOps**: one team builds, tests, deploys and runs small services, released often
- **"It works on my machine"** → ship the app *with* its environment
- **Container**: app + bins/libs, as normal processes on a shared host kernel
- **Image vs container**: read-only layers, built once vs a running instance + a writable layer
- **Containers vs VMs**: no guest kernel, lighter and faster; VMs isolate more (and often host the containers)

→ Next time: Containers in Practice lab Modules 2–4

engines, images, volumes, networks, builds, Compose

Containers 2/3: Containers in Practice

Engines, images, storage, networking, builds and Compose

Walter Rudametkin

Walter.Rudametkin@univ-rennes.fr

https://rudametw.github.io/teaching

ESIR 2 SI — Architectures Logicielles

Université de Rennes







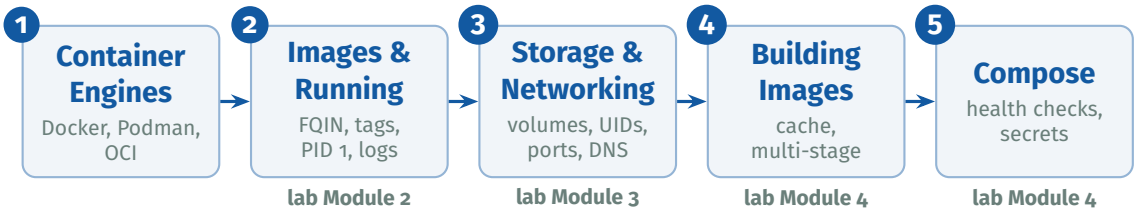






Lecture 2: Road Map

1. DevOps and Why Containers	2. Containers in Practice	3. Under the Hood and Production
lab Module 0	today · lab Modules 2–4	lab Modules 0–1 and 5



Goal: usage first: what you ran in the lab, and why.

Previously: DevOps, why containers, containers vs VMs (lecture 1).

appendix: deep dives **field guide**: recipes, troubleshooting

Build, Ship, Run



Dockerfile
(Build)

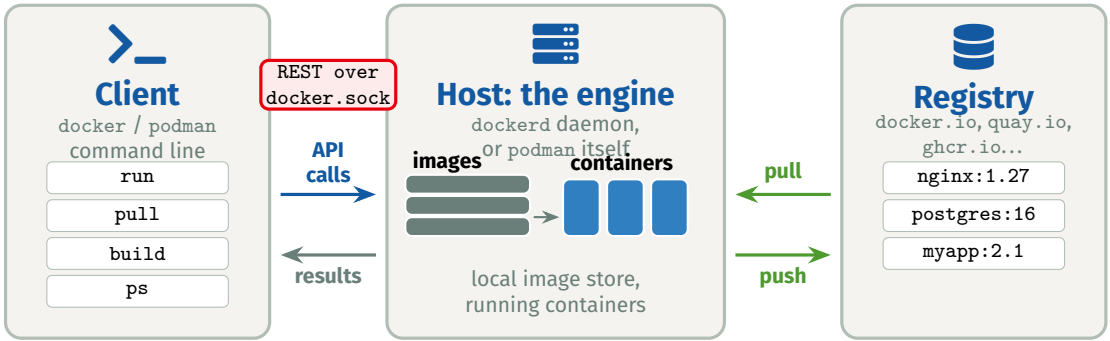


Docker Image
(Ship)



Containers
(Run)

Client → Engine → Registry



Docker: `dockerd`, a **root** daemon (rootless mode: opt-in), serves this API on `/var/run/docker.sock`.

⚠️ Access to `docker.sock` (`docker` group) ≈ root on the host.

Podman: client and engine in one program, no daemon.

The Open Container Initiative (OCI)

Founded in 2015 (Docker, CoreOS, Google, Red Hat…) to standardize:

- **Image spec**: layers (tarballs) + JSON config + manifest
  - a "Docker image" is an OCI image
- **Runtime spec**: how to run a "filesystem bundle"
  - `config.json`: process, mounts, namespaces, cgroups, capabilities…
  - low-level OCI runtimes: `runc`, `crun` (details: appendix)
- **Distribution spec**: the HTTP API of registries (push/pull)

✔ Docker, Podman, Kubernetes…all pull and run *the same images*.

Podman

Daemonless

- no background service
- runs `conmon` + `crun`
- children of *your* session

Rootless by default

- runs as an unprivileged user
- “root” inside = you outside

Docker-compatible

- same OCI images, same CLI
- optional Docker API socket

Beyond Docker

- pods, `podman kube play`
- systemd integration: Quadlets
- GUI: Podman Desktop

```
$ systemctl --user enable --now podman.socket    # Docker API
$ export DOCKER_HOST=unix://$XDG_RUNTIME_DIR/podman/podman.sock
```

Then tools written for Docker (`docker-compose`, IDEs, Testcontainers) drive Podman.

Docker ≡ Podman: Everyday Commands

Docker	Podman	Purpose
`docker pull`	`podman pull`	download an image
`docker images`	`podman images`	list local images
`docker run`	`podman run`	create + start a container
`docker ps -a`	`podman ps -a`	list containers (incl. exited)
`docker logs`	`podman logs`	stdout/stderr of PID 1
`docker exec -it`	`podman exec -it`	run a command inside
`docker inspect`	`podman inspect`	full JSON config/state
`docker rm -f`	`podman rm -f`	kill + remove a container
`docker build`	`podman build`	build an image
`docker compose`	`podman compose`*	multi-container stacks
—	`podman unshare`	enter the rootless user ns.
—	`podman machine`	Linux VM (macOS/Windows)

* a wrapper: runs `podman-compose` or `docker-compose`.

Our First Containers

```
$ podman run docker.io/library/busybox:1.36 echo hello world
hello world
$ podman run -it docker.io/library/ubuntu:24.04
root@04c0bb0a6c07:/# apt-get update && apt-get install -y figlet
root@04c0bb0a6c07:/# figlet hello; exit
$ podman run -it docker.io/library/ubuntu:24.04     # again
root@b13c164401fb:/# figlet
bash: figlet: command not found
```

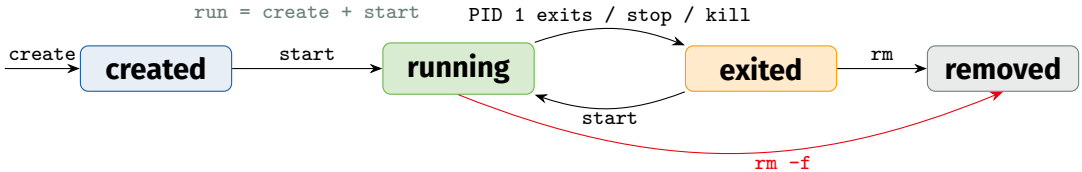
- `run` = pull (if missing) + create + start + attach; `-it` = stdin + a terminal
- each `run` = a new container: `figlet` stayed in the *old*, stopped one
- need `figlet`? build an image: containers are cattle, not pets

Background Containers, Names and Logs

```
$ podman run -d --name ticker docker.io/library/alpine:3.20 \
      sh -c 'while true; do date; sleep 1; done'
$ podman ps               # running containers only
$ podman ps -a            # all, including exited ones
$ podman logs --tail 3 -f ticker
$ podman run --rm -it docker.io/library/alpine:3.20 sh
```

- `-d` (*detached*): prints the container ID and returns
- the engine keeps stdout/stderr of PID 1 → `podman logs`
- `--name`: otherwise a random name (`nostalgic_turing`)
- `--rm`: remove the container when it exits

Container Lifecycle: It Lives as Long as Its PID 1



- PID 1 (`Entrypoint` + `Cmd`) exits ⇒ all its processes are killed
- servers run in the foreground; `ubuntu` without `-it`: `bash` reads EOF → `Exited (0)`
- `stop`: `SIGTERM`, `SIGKILL` after 10 s; `kill`: `SIGKILL`; `start`: the same container
- exit `0` OK, `1` app error, `137` = 128+`SIGKILL` (kill, OOM); `--init`: tiny init as PID 1

Lab: Module 2, Ex. 2.2
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

Fully Qualified Image Names (FQIN)

Registries serve images: `docker.io` (Docker Hub; `library/` = official images), `quay.io`, `ghcr.io`, private ones…



Short names are a trap: what is `postgres`?

- **Docker**: silently `docker.io/library/postgres:latest`
- **Podman**: a known alias (`000-shortnames.conf`), else a prompt in a terminal, an error in scripts
- whoever owns that name on a searched registry wins

Rule: always write the full name.

Lab: Module 2, Conceptual foundation

Tags Are Mutable, Digests Are Not: Pin Your Versions

Git	Images	
branch	tag `:latest`, `:16`, `:16-alpine`	moves with each release
tag	tag `:16.4-alpine`	should not move (but can!)
commit hash	digest `@sha256:4f6e...`	immutable: hash of the manifest

- a tag is a pointer the maintainer can move; `latest` = the default name, not "the newest"
- risk: `postgres:latest` is now 18, with a new data directory: your old volume no longer matches
- pin: `postgres:16.4-alpine` (production), `postgres@sha256:...` (reproducible builds)
- digest: `podman image inspect --format '{{.Digest}}' IMG`; upgrade on purpose

Lab: Module 2, Conceptual foundation

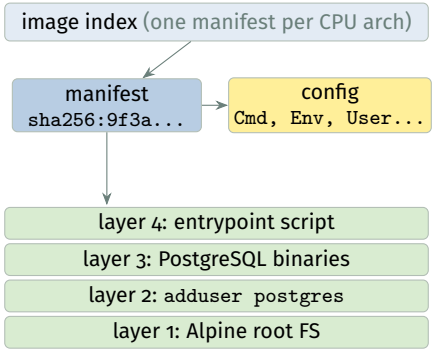
What Is a Container Image, Really?

Not a VM disk, but:

- layers: `tar`s of file changes
- a config: what to run, and how
- a manifest: lists them by `sha256`

Standard format: the OCI image spec

See the layers: `podman image tree`, `podman history` (details: appendix)



stacked by OverlayFS (lecture 3)

https://github.com/opencontainers/image-spec

Auditing an Image: The Fields to Check

```
$ podman image inspect --format '{{json .Config}}' \
      docker.io/library/postgres:16-alpine | jq .
```

Config field	Containerfile	go-httpbin	postgres
`ExposedPorts`	`EXPOSE`	`8080/tcp`	`5432/tcp`
`Volumes`	`VOLUME`	—	`/var/lib/postgresql/data`
`User`	`USER`	`0` (root)	empty: root, then UID 70
`Entrypoint`	`ENTRYPOINT`	—	`docker-entrypoint.sh`
`Cmd`	`CMD`	`/bin/go-httpbin`	`postgres`
`Env`	`ENV`	`SSL_CERT_FILE`	`PGDATA`, `PG_MAJOR`
`StopSignal`	`STOPSIGNAL`	(`SIGTERM`)	`SIGINT`

`ExposedPorts` → your `-p`; `Volumes` → `-v`; `User` → file owners; `Entrypoint` + `Cmd` = PID 1 (`podman run IMG a b` replaces `Cmd`). All overridable (`--user`, `--entrypoint`…).

Lab: Module 2, Ex. 2.1

Diagnosing a Container That Exits Immediately

```
$ podman run -d --name broken-db docker.io/library/postgres:16-alpine
$ podman ps                                          # empty!
$ podman ps -a --format '{{.Names}} {{.Status}}'
broken-db Exited (1) 3 seconds ago
$ podman logs broken-db
Error: Database is uninitialized and superuser password is not
       specified. You must specify POSTGRES_PASSWORD...
```

- `podman ps` hides exited containers: check `ps -a`
- `logs` works on stopped containers: first reflex when it fails

Resource limits (`--memory`, `--cpus`…): appendix; their cgroups: lecture 3.

Lab: Module 2, Ex. 2.2

Configuring with `-e`, Entering with `exec`

```
$ podman rm broken-db
$ podman run -d --name working-db \
      -e POSTGRES_USER=esir -e POSTGRES_PASSWORD=secretpass \
      -e POSTGRES_DB=coursedb \
      docker.io/library/postgres:16-alpine
$ podman exec -it working-db psql -U esir -d coursedb
$ podman exec -it working-db sh        # interactive shell
$ podman diff working-db               # files changed vs image
$ podman rm -f working-db              # done with it: kill + remove
```

- `-e KEY=VALUE`: environment of PID 1; shown by `podman inspect` (secrets: see Compose)
- `exec`: a new process in the same namespaces and cgroup; no SSH needed
- only the image's tools: *distroless* has no shell

Lab: Module 2, Ex. 2.2

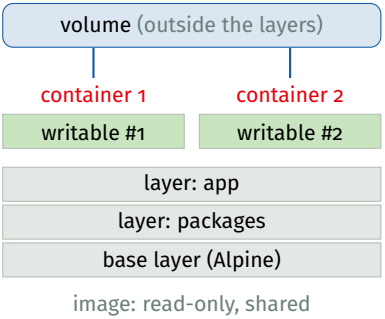
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

The Writable Layer Is Ephemeral

- writes go to a thin writable layer (OverlayFS)
- `podman rm` ⇒ it is deleted
- copy-on-write is slow for databases

⇒ **keep data *outside***

- named volumes (engine-managed)
- bind mounts (host directory)



Named Volumes and Bind Mounts

```
$ podman run -d -e POSTGRES_PASSWORD=secretpass \
      -v pgdata:/var/lib/postgresql/data \
      docker.io/library/postgres:16-alpine          # named volume
$ podman run -d -v ~/site:/usr/share/nginx/html:ro,Z \
      docker.io/library/nginx:1.27                  # bind mount
```

`-v SRC:DST[:OPTS]`: `SRC` without `/` = a volume name, otherwise a host path

Named volume (`pgdata`)

- engine-managed: `podman volume ls` / `inspect` / `rm`
- survives `podman rm`
- filled from the image (content + owner): rootless-friendly
- databases, app state

Bind mount (`~/site`)

- any host path, edited from the host
- host owners and modes as is → rootless trap (next)
- SELinux: `:Z` (private), `:z` (shared); `:ro`: read-only
- source code, config files

Rootless volumes live in `~/.local/share/containers/storage/volumes/`

Rootless Storage: The Permission Trap

Symptom: writing into a bind-mounted directory → `Permission denied` (e.g. `redis` as UID 999, in `/data`)

Cause: container UID 999 = host UID $100000 + 999 - 1 = 100998$ (`/etc/subuid`: `me:100000:65536`); the directory belongs to you (1000) = container `root`: 100998 is "others" (`r-x`)

Three fixes

```
$ podman run -v ~/redis-bind:/data:U ...     # 1. Podman chowns it
$ podman unshare chown 999:999 ~/redis-bind  # 2. chown in your userns
$ podman run -v redis-data:/data ...         # 3. named volume
```

SELinux (Fedora, RHEL): add `:Z` (`:z` if shared): `-v ~/redis-bind:/data:Z,U`

Worked example: appendix.

Lab: Module 3, Ex. 3.1

Publishing Ports

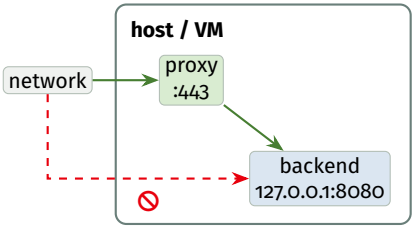
```
$ podman run -d --name web -p 8080:80 docker.io/library/nginx:1.27
$ curl http://localhost:8080/
$ podman port web
80/tcp -> 0.0.0.0:8080
```

- `-p HOST:CONTAINER`: host port 8080 → container port 80
- without `-p`: unreachable from outside the host
- `EXPOSE` / `ExposedPorts`: metadata only, publishes nothing…
- …except `-P`: *all* exposed ports, on random host ports
- rootless: host ports < 1024 are privileged

Lab: Module 3, Ex. 3.2

Port Binding Security: `0.0.0.0` vs `127.0.0.1`

- `-p 8080:8080` binds all interfaces (`0.0.0.0`): on a shared VM, your backend is public
- behind a local reverse proxy, bind to loopback only




```
$ podman run -d --name http-echo -p 127.0.0.1:8080:8080 \
      docker.io/mccutchen/go-httpbin:v2.15.0
$ curl 127.0.0.1:8080/ip && podman port http-echo
```

Firewall: rootless Podman (pasta) ports are ordinary sockets, the host firewall applies; rootful Docker's own `iptables` rules bypass `ufw`/`firewalld`.

Lab: Module 3, Ex. 3.2

User-Defined Networks and DNS

```
$ podman network create esir-net
$ podman network inspect esir-net     # subnet 10.89.0.0/24, DNS on
$ podman run -d --name backend-api --network esir-net \
      docker.io/mccutchen/go-httpbin:v2.15.0          # no -p!
$ podman run --rm --network esir-net \
      docker.io/library/alpine:3.20 \
      wget -qO- backend-api:8080/headers
{ "headers": { "Host": ["backend-api:8080"], ... } }
```

- the container name is its DNS name (not on the default network; replaces `--link`)
- reachable on its network, invisible from outside the host

Podman: netavark (bridges, firewall) + aardvark-dns; rootless: a private namespace linked out by pasta, container IPs unreachable from the host (appendix)

Lab: Module 3, Ex. 3.2

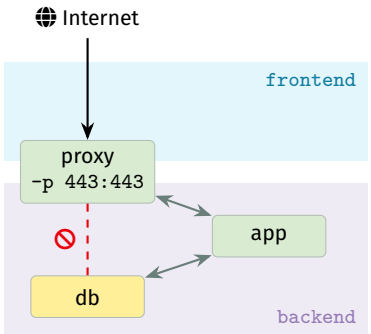
Networks: Who Can Talk to Whom

Simple rule: containers talk only if they share a network.

Segmentation

- one network per role or per stack
- only the reverse proxy publishes ports
- backends, databases: no `-p`
- a hacked proxy cannot reach the DB

(Compose does this for you)



Lab: Module 3, Ex. 3.2

Containerfile (a.k.a. Dockerfile)

A **build recipe**, run top to bottom. `Containerfile` = `Dockerfile` (same syntax); `podman build -t myimage:1.0 .` ≡ `docker build`

- each `RUN`, `COPY`, `ADD` = an immutable layer (snapshot)
- `ENV`, `CMD`, `EXPOSE`…: metadata only (0 B)
- layers are **cached and reused** between builds

```
$ podman history esir-app:naive                (shortened)
CREATED BY                                     SIZE
/bin/sh -c #(nop) CMD ["/app"]                 0 B
/bin/sh -c go build -o /app main.go            7.8 MB
/bin/sh -c #(nop) COPY file:... .              1.5 kB
(golang:1.23-alpine layers)                    250 MB
```

Anti-pattern: set up a container by hand + `commit`: not reproducible.

Lab: Module 4, Ex. 4.1

Build, Run, Push, Pull

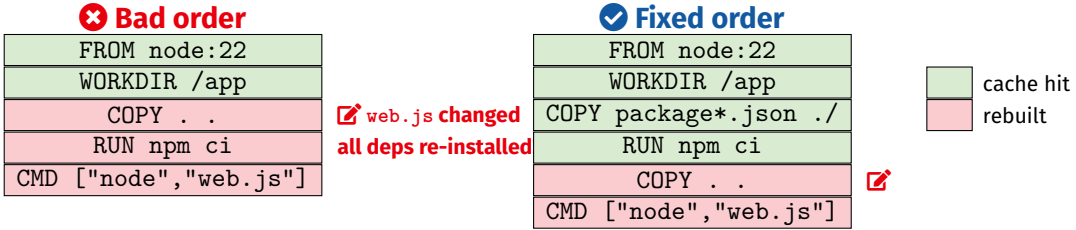
```
# Build an image from ./Containerfile (or ./Dockerfile)
podman build -t quay.io/esir/web:1.0 .
# Run it: host 127.0.0.1:8080 -> container port 3000
podman run -d -p 127.0.0.1:8080:3000 quay.io/esir/web:1.0
# Push to a registry                              (developer)
podman login quay.io
podman push quay.io/esir/web:1.0
# Pull it anywhere else                           (operations)
podman pull quay.io/esir/web:1.0
```

The image name says where it is pushed (`podman tag SRC NEW` adds one); no registry prefix: `-t web:1.0` → `localhost/web:1.0`.

Docker: same sub-commands.

The Build Cache: Order Matters

A layer is **reused** if same parent + same instruction: `RUN` compares the text only (`apt-get update` is not re-run!), `COPY`/`ADD` a **checksum of the files**. After the first miss, every following step is rebuilt. (`--no-cache`: rebuild all.)



Dependency manifests first (`package*.json`, `go.mod`, `pom.xml`…), source code last.

(`node:22` = `docker.io/library/node:22`)

Lab: Module 4, Ex. 4.1

Deleting a File Later Does Not Shrink the Image

Instruction	Layer size	Image size
`FROM docker.io/library/debian:13-slim`	base	base
`RUN apt-get install -y gcc`	+200 MB	base + 200 MB
`RUN apt-get purge -y gcc`	≈ 0 (whiteouts)	base + 200 MB

Lower layers are **immutable**: a deletion only adds whiteouts. The files are still shipped and extractable: a secret `COPY`'d then `rm`'d is still there. `RUN chown`/`chmod`/`mv`: the whole file is copied again.

Fix 1: create and delete in the same `RUN`

```
RUN curl -fsSLO https://example.org/foo.tgz \
 && tar -xzf foo.tgz -C /usr/local \
 && rm foo.tgz
```

Fix 2: multi-stage builds

build in a "fat" stage, `COPY` only the result: build tools never enter the final image

Lab: Module 4, Ex. 4.1

Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

The Naive Build (Anti-Pattern)

`main.go`: a tiny web server (Go standard library; port from `$PORT`; code: appendix).
`Containerfile.naive`:

```
FROM docker.io/library/golang:1.23-alpine
WORKDIR /src
COPY main.go .
RUN go build -o /app main.go
EXPOSE 3000
CMD ["/app"]
```

```
$ podman build -t esir-app:naive -f Containerfile.naive .
```

It works, but the image:

- weighs more than 250 MB for a program of a few MB
- ships the Go compiler, a shell, `apk`; runs as root (no `USER`)

Lab: Module 4, Ex. 4.1

The Production Multi-Stage Build

Each `FROM` starts a stage; only the last one becomes the image.

```
# Stage 1: build environment, discarded afterwards
FROM docker.io/library/golang:1.23-alpine AS builder ← stage name
WORKDIR /src
COPY main.go .
RUN CGO_ENABLED=0 GOOS=linux \      ← pure Go: a static binary, no C library
    go build -ldflags="-s -w" -o /app main.go ← strip debug info
# Stage 2: minimal runtime image
FROM docker.io/library/alpine:3.20      ← ≈ 8 MB base (still a shell + apk)
RUN adduser -D -u 10001 appuser         ← unprivileged account, no password
USER 10001                              ← numeric UID: no /etc/passwd lookup
COPY --from=builder /app /usr/local/bin/app ← only the binary
ENV PORT=3000
EXPOSE 3000
ENTRYPOINT ["/usr/local/bin/app"]       ← exec form: PID 1, gets SIGTERM
```

Lab: Module 4, Ex. 4.1

Compare the Two Images: Size and Attack Surface

```
$ podman build -t esir-app:prod -f Containerfile.prod .
$ podman images esir-app                        (shortened)
REPOSITORY          TAG     SIZE
localhost/esir-app  prod    ~10MB
localhost/esir-app  naive   >250MB
```

	`:naive`	`:prod`	**With an RCE bug, an attacker…**
Size	> 250 MB	≈ **10 MB**	(pull time, disk, CVEs to patch)
Go compiler + sources	yes	no	naive: uses the bundled compiler
Shell, package manager	yes	yes (Alpine)	opens a shell, `apk add`s miners…
Runs as	root	**UID 10001**	prod: can't install or alter system files

Rootless: UID 10001 is an **unprivileged subuid** on the host. Further: `gcr.io/distroless/static` (no shell) or `FROM scratch`.

Lab: Module 4, Ex. 4.1

Containerfile Checklist

- ✔ **Pin** FQ base images (tag, or better, digest)
- ✔ **Dependency manifests first**, source code last
- ✔ `update && install && cleanup` in one `RUN`
- ✔ No secrets in `COPY`, `ARG`, `ENV` (history shows them)
- ✔ **Multi-stage**: build tools stay behind
- ✔ Run as a **non-root**, numeric `USER`
- ✔ **Exec form** for `ENTRYPOINT`/`CMD`
- ✔ `COPY --chown=10001`, not `RUN chown`
- ✔ Configure via `ENV`, override with `-e`

Start simple, optimize when it matters. All instructions, build secrets, signing: appendix.

Build context: the `.` in `podman build -t web:1.0 .`
the only files `COPY`/`ADD` see, sent to the builder; `-f`: another Containerfile

`.containerignore` (or `.dockerignore`)

```
.git
node_modules
*.log
.env
```

`.gitignore` syntax: faster builds, secrets stay out of the image

Compose

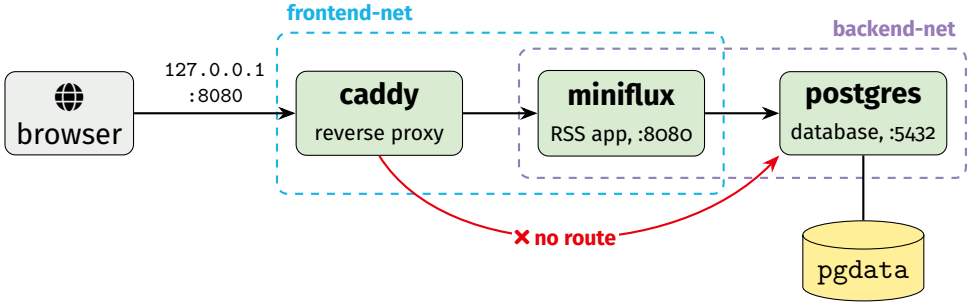
One container = one concern (one main process): an app = several containers, each with its own options, connected together.

Compose: define and run **multi-container applications** on one host

- one YAML file, `compose.yaml`: `services`, `networks`, `volumes`
- service keys = `podman run` options (`ports`: `-p`, `volumes`: `-v`…)
- **declarative**: `podman compose up` converges to the file
- open standard: the **Compose Specification**
  - `docker compose` (Go plugin)
  - `podman compose` (wrapper: `podman-compose` or `docker-compose`)

`version:` is obsolete. The *project name* (directory name) prefixes everything: `selfhosted-stack_pgdata`.

Lab Capstone: A Self-Hosted RSS Reader



- Only `caddy` publishes a port, and only on **loopback**
- `caddy` cannot reach `postgres`: they share no network
- Data lives in the **named volume** `pgdata`

Lab: Module 4, Ex. 4.2

Configuration Next to `compose.yaml`

`Caddyfile`

```
:8080 {
  reverse_proxy miniflux:8080
}
```

- listen on 8080 (rootless: port > 1024)
- forward to `miniflux` **by name** (network DNS)
- the place for TLS, auth, rate limits

`.env` (read by Compose)

```
DB_USER=miniflux
DB_PASSWORD=<random secret>
DB_NAME=miniflux
ADMIN_USER=admin
ADMIN_PASS=<another secret>
```

```
chmod 600 .env     # owner only
```

- `${DB_USER}` in `compose.yaml` ← value
- never commit it (`.gitignore`)
- values still visible: see *Secrets*

Lab: Module 4, Ex. 4.2

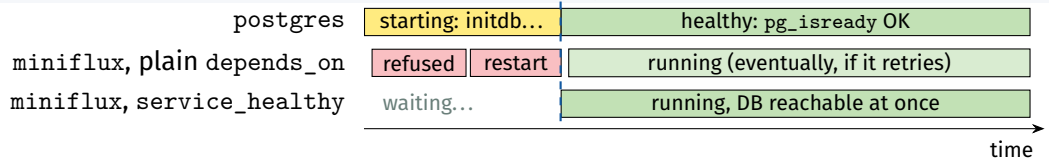
`compose.yaml` (1/4): The Database

```yaml
services:
  postgres:
    image: docker.io/library/postgres:16-alpine
    container_name: stack-postgres
    restart: unless-stopped                # restart policy
    environment:                           # values from .env
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: ${DB_NAME}
    volumes:
      - pgdata:/var/lib/postgresql/data    # NAMED volume
    networks:                              # no "ports:" at all
      - backend-net
    mem_limit: 256m                        # cgroup memory.max
```

Lab: Module 4, Ex. 4.2

`compose.yaml` (2/4): Health Checks, Started vs Healthy

```yaml
postgres:
  healthcheck:      # runs INSIDE the container; exit 0 = healthy
    test: ["CMD-SHELL", "pg_isready -U ${DB_USER} -d ${DB_NAME}"]
    interval: 5s    # run every 5 s
    timeout: 3s     # slower = a failure
    retries: 5      # 5 in a row = unhealthy (podman ps)
```



Running ≠ ready: `depends_on` alone only waits for started.

Containerfile `HEALTHCHECK`: dropped by `podman build` (OCI) unless `--format docker`. Declare it here, or Quadlet `HealthCmd=`.

Lab: Module 4, Ex. 4.2

`compose.yaml` (3/4): The Application

```yaml
  miniflux:
    image: docker.io/miniflux/miniflux:latest   # pin it in prod
    depends_on:
      postgres:
        condition: service_healthy      # wait for pg_isready
    environment:
      DATABASE_URL:
    ↪ postgres://${DB_USER}:${DB_PASSWORD}@postgres/${DB_NAME}?sslmode=disable
      RUN_MIGRATIONS: "1"      # + CREATE_ADMIN, ADMIN_USERNAME...
    networks:
      - backend-net                     # to reach postgres
      - frontend-net                    # to be reached by caddy
    mem_limit: 128m
```

`@postgres`: the **service name**, resolved by the network's DNS (details: appendix)

Lab: Module 4, Ex. 4.2

`compose.yaml` (4/4): The Proxy, Volumes, Networks

```yaml
  caddy:
    image: docker.io/library/caddy:2-alpine
    depends_on:
      - miniflux            # only "started"
    ports:
      - "127.0.0.1:8080:8080"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile:ro,Z
    networks:
      - frontend-net
    mem_limit: 64m
```

```yaml
volumes:
  pgdata:

networks:
  frontend-net:
  backend-net:
  # internal: true
```

`127.0.0.1:8080:8080`: the **only** published port, loopback only.
`:ro,Z`: read-only **bind mount**, relabelled for SELinux.
`internal: true` (optional): `backend-net` gets no route out.

Lab: Module 4, Ex. 4.2

Running the Stack

```
cd ~/lab-data/selfhosted-stack
podman compose up -d                # networks, volume, containers
podman compose ps                   # this project's containers
podman compose logs -f miniflux     # follow one service
curl -i http://127.0.0.1:8080/login    # through caddy
podman compose up -d    # after an edit: re-creates changed services*
podman compose down     # stop + remove containers and networks
```

`restart:` when PID 1 exits

`no` (default), `on-failure`, `always`, `unless-stopped`. Needs a watcher: `dockerd`, `conmon` (Podman)… while your session lives (lecture 3)

`mem_limit:` ⇒ cgroup v2 `memory.max`

Over it: OOM killer (exit 137 = 128 + 9); a runaway service cannot starve the VM (also `cpus:`, `pids_limit:`)

* Provider-dependent: check `podman compose version`.

Lab: Module 4, Ex. 4.2

What Survives `compose down`?

	`down`	`down -v`
Containers	removed	removed
Networks of the project	removed	removed
Named volumes (`pgdata`)	**kept**	removed
Bind-mounted files (`./Caddyfile`)	kept	kept
Images	kept	kept

- Containers are **disposable**; the state lives in the **volume**
- `down` then `up -d`: new containers, same data
- `down -v` is the "factory reset": the database is gone

Lab: Module 4, Ex. 4.2
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

Secrets: Keep Them Out of the Environment

Environment variables (`-e`, `environment:`, `.env` values) are easy to read:

```
$ podman inspect stack-postgres --format '{{.Config.Env}}'
[... POSTGRES_PASSWORD=s3cr3t ...]          # also in /proc/1/environ
$ printf '%s' "$DB_PASSWORD" | podman secret create db_pw -
$ podman run --secret db_pw ...     # a FILE: /run/secrets/db_pw
```

Compose: a `secrets:` file, and the image's `_FILE` variant

```
  postgres:
    secrets: [db_pw]
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_pw
secrets:
  db_pw: {file: ./db_pw.txt}        # chmod 600, not in git
```

Quadlet (lecture 3): `Secret=db_pw,type=env,target=POSTGRES_PASSWORD` (hidden from `inspect`)

`.env` + `chmod 600` only hides the *file* from other users. Recipes: the field guide.

Review This File: Find 4 Problems

```yaml
services:
  redis:
    image: 'redislabs/redismod'
    ports:
      - '6379:6379'
  web1:
    restart: on-failure
    build: ./web
    hostname: web1
    ports:
      - '81:5000'
```

```yaml
  web2:
    restart: on-failure
    build: ./web
    hostname: web2
    ports:
      - '82:5000'
  nginx:
    build: ./nginx
    ports:
    - '80:80'
    depends_on:
    - web1
    - web2
```

First: which services are deployed? What is `nginx` for?

Review: The Answers

Reading it: 4 services; `web1`, `web2`: the same image (`./web`); `nginx`: a **load balancer** (custom config) reaching them as `http://web1:5000` (DNS)

1. `redislabs/redismod`: short name, no tag (`:latest`) → FQIN, pinned version
2. every port published on `0.0.0.0`, Redis included → only the proxy needs `ports:`; separate networks
3. port 80: refused to rootless Podman (< 1024) → e.g. `127.0.0.1:8080:80`
4. plain `depends_on`: started, not ready → `healthcheck` + `condition: service_healthy`

Also: no volume for Redis data, no `mem_limit`.

Lecture 2: What to Remember

- **Images**: FQIN + pinned tags (or digests); a container lives as long as its PID 1
- **Data**: the writable layer is ephemeral → volumes; rootless permission trap → `:U`, `:Z`
- **Networking**: publish on `127.0.0.1`; user-defined networks + DNS
- **Builds**: cache order, multi-stage, non-root `USER`
- **Compose**: one file per stack, health checks, secrets out of the environment

→ Next time: Under the Hood and Production lab Modules 0–1 and 5

namespaces, cgroups, security, OverlayFS; linger, Quadlets (systemd services)

Containers 3/3: Under the Hood and Production

Linux kernel primitives; rootless containers as systemd services

Walter Rudametkin

Walter.Rudametkin@univ-rennes.fr

https://rudametw.github.io/teaching

ESIR 2 SI — Architectures Logicielles

Université de Rennes













Lecture 3: Road Map

1. DevOps and Why Containers
lab Module 0

2. Containers in Practice
lab Modules 2–4

3. Under the Hood and Production
today · lab Modules 0–1 and 5



Goal: understand what `podman run` *really* does, and run a stack that survives logout and reboot.

Previously: images, volumes, networks, builds, Compose (lecture 2).

appendix: deep dives **field guide**: recipes, troubleshooting

There Is No “Container” in the Linux Kernel

No “container” object in the kernel: the engine (Podman, Docker) only *configures* **independent kernel features**:

Primitive	Responsibility	Kernel interface
Namespaces	what it *sees*: PIDs, mounts, net, users	`clone`, `unshare`, `setns` (2)
cgroups v2	what it *uses*: CPU, RAM, I/O, PIDs	files in `/sys/fs/cgroup`
Security	what it *may do*: capabilities, seccomp, LSM	`capset`, `seccomp` (2), SELinux
OverlayFS	how its files are *stored*: layers, copy-on-write	`mount -t overlay`

A container is just an ordinary Linux process
started with these features configured.

Lab: Module 1, introduction
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

A Container Is Just a Process on the Shared Kernel

```
$ podman run -d --name sleeper docker.io/library/alpine:3.20 sleep 600
$ ps -o user=,pid=,cmd= -C sleep        # seen from the host
alice    24817 sleep 600
$ readlink /proc/24817/ns/uts /proc/self/ns/uts
uts:[4026534010]
uts:[4026531838]
$ uname -r; podman exec sleeper uname -r
6.12.4-200.fc41.x86_64
6.12.4-200.fc41.x86_64
```

Just a process

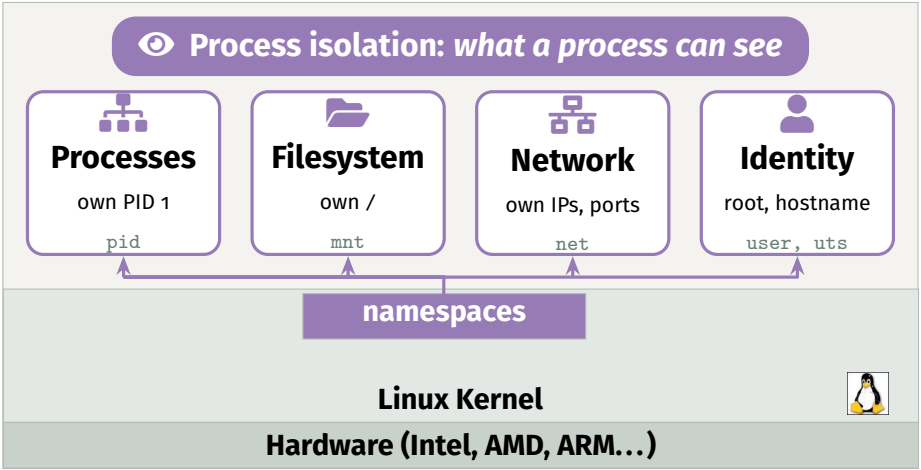
a host process, owned by you, in *other namespaces* (other inode numbers)

One shared kernel

image = *userland files* only: nothing to boot (ms), but a kernel bug hits all containers

Lab: Module 0; Module 1, Ex. 1.1 (step 6)

Linux Container Architecture: Namespaces



also: `ipc`, `cgroup` and `time` namespaces

Why Namespaces? What Each One Isolates

Problems: two apps on port 80 · an app kills all processes · "root" without being root
A namespace = a process's own view of one kind of resource:

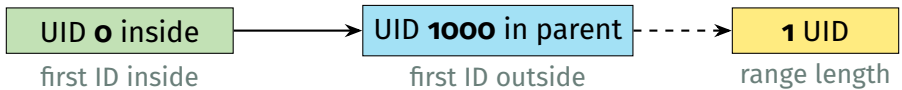
Namespace	Isolates	In a container
`mnt`	mounts, root filesystem	own `/` (image), via `pivot_root(2)`
`uts`	hostname	`podman run --hostname web`
`ipc`	System V IPC, POSIX queues	no shared memory between containers
`pid`	process IDs	entrypoint = PID 1 (own `/proc`)
`net`	interfaces, routes, firewall	own `lo`; each can listen on `:80`
`user`	UIDs/GIDs, capabilities	"root" inside = your UID outside
`cgroup`	view of the cgroup tree	`/sys/fs/cgroup` = its own node
`time`	monotonic, boot-time clocks	checkpoint / restore (CRIU)

One namespace of each type per process: `ls -l /proc/self/ns` (appendix)

User Namespace and `uid_map`

- Maps UIDs/GIDs inside ↔ parent namespace
- UID 0 inside: privileged *on the namespace's own resources* (mounts, interfaces, hostname); outside: an unprivileged user
- Linux 3.8 (2013): any user can create one → rootless containers

```
$ unshare --user --map-root-user bash
# cat /proc/self/uid_map
         0       1000          1
```



Unmapped UIDs show as `nobody` (65534); files of "root" inside are *yours* on the host.

Lab: A Hand-Made Sandbox with `unshare`

```
$ id -u; echo $$                          # 1000, and e.g. 4242
$ unshare --user --map-root-user --pid --mount --uts --fork bash
```

`--user`	new user namespace (`CLONE_NEWUSER`), created first
`--map-root-user`	write `0 1000 1` in `uid_map`: you are root inside
`--pid`	new PID namespace
`--mount`	new mount namespace (private mounts)
`--uts`	new hostname
`--fork`	fork before `exec`, so that `bash` is PID 1

No `sudo`, no Podman, no Docker: only the kernel.

Lab: Module 1, Ex. 1.1

Lab: Inside the Sandbox

```
# id
uid=0(root) gid=0(root) ...
# hostname esir-sandbox
# hostname
esir-sandbox
# echo $$
1
# ps aux | wc -l
312
```

Root, PID 1, own hostname…
but every host process?!

```
# mount -t proc proc /proc
# ps aux
USER  PID ... COMMAND
root    1 ... bash
root    2 ... ps aux
```

- `ps` reads `/proc`: still the parent's mount
- New `proc` mount = *our* PID namespace
- Allowed by `--mount` + `--map-root-user`

Lab: Module 1, Ex. 1.1 (steps 3–4)

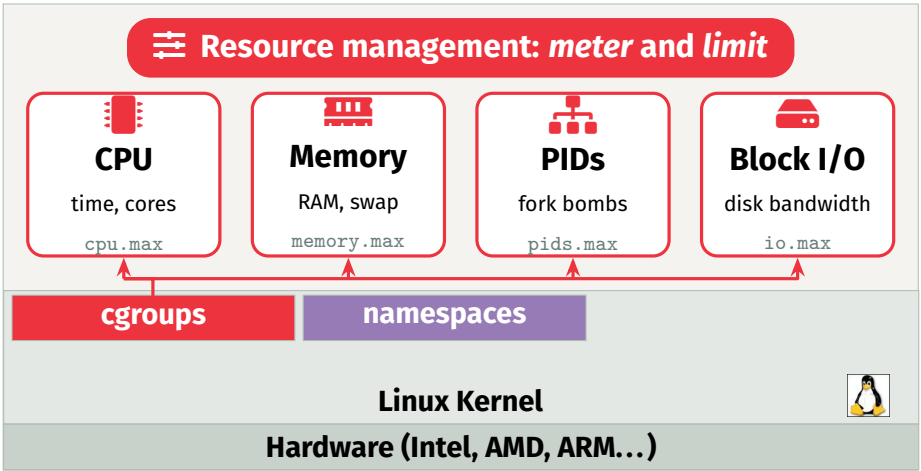
Rootless: `/etc/subuid`, `newuidmap`, Sanity Check

- Images hold files of many users: `alice:100000:65536` reserves host UIDs 100000–165535; setuid `newuidmap` / `newgidmap` write the map
- UID 0 → 1000 (you); UID $n \geq 1 \rightarrow 100000 + n - 1$ (example: lecture 2, appendix)

```
$ podman info --format \
    '{{.Host.Security.Rootless}} | {{.Host.CgroupsVersion}}'
true | v2                        note the s (Podman 5)
$ grep $(whoami) /etc/subuid
alice:100000:65536               missing?  usermod --add-subuids
$ podman unshare cat /proc/self/uid_map
         0       1000          1
         1     100000      65536
```

Lab: Module 0, sanity check; Module 3

Linux Container Architecture: cgroups



also: `cpuset`, `hugetlb`, devices (eBPF filter); no network controller in v2

Why cgroups? What Can They Control?

- Namespaces *hide*, they do not *limit*: one memory leak, busy loop or fork bomb can starve the whole host
- cgroups (Linux 2.6.24, 2008): group processes, then meter and limit their memory, CPU, block I/O, PIDs, devices…
- **Crowd control**: freeze / thaw a group (`cgroup.freeze`, `podman pause`), kill it with no fork escape (`cgroup.kill`)
- systemd: one cgroup per service and session; engines: one per container

Lab: Module 1, Ex. 1.3
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

The cgroup v2 Filesystem

```
$ stat -fc %T /sys/fs/cgroup          # tmpfs would mean v1
cgroup2fs
$ cat /sys/fs/cgroup/cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
$ cat /proc/self/cgroup                # which node am I in?
0::/user.slice/user-1000.slice/user@1000.service/app.slice/...
$ ls /sys/fs/cgroup/user.slice/
cgroup.procs  cpu.max  io.max  memory.current  memory.max  pids.max ...
```

- A **tree**: `mkdir` = new node; files = knobs (`memory.max`), counters, members (`cgroup.procs`); ancestors' limits apply too
- Rootless containers sit in your `user-1000.slice`: they die with your session unless "lingering"

v2 (Linux 4.5, 2016): one unified tree, *delegated* to users (rootless); v1: appendix

Lab: Module 1, Ex. 1.3 (step 1)

Lab: A 16 MiB Container

```
$ podman run --rm -it --name mem-test --memory=16m \
      --memory-swap=16m docker.io/library/alpine:3.20 sh
/ # cat /sys/fs/cgroup/memory.max
16777216                                  = 16 x 1024 x 1024 bytes
/ # cat /sys/fs/cgroup/memory.swap.max
0                                         --memory-swap = RAM + swap
/ # tail /dev/zero
Killed
/ # echo $?
137
/ # grep oom_kill /sys/fs/cgroup/memory.events
oom_kill 1
```

Inside the container, `/sys/fs/cgroup` *is* the container's cgroup (cgroup namespace).

Lab: Module 1, Ex. 1.3

Memory Limits, the OOM Killer and Exit 137

`memory.current`	bytes used now
`memory.high`	throttle, *no* kill
`memory.max`	hard limit
`memory.events`	`oom`, `oom_kill`…

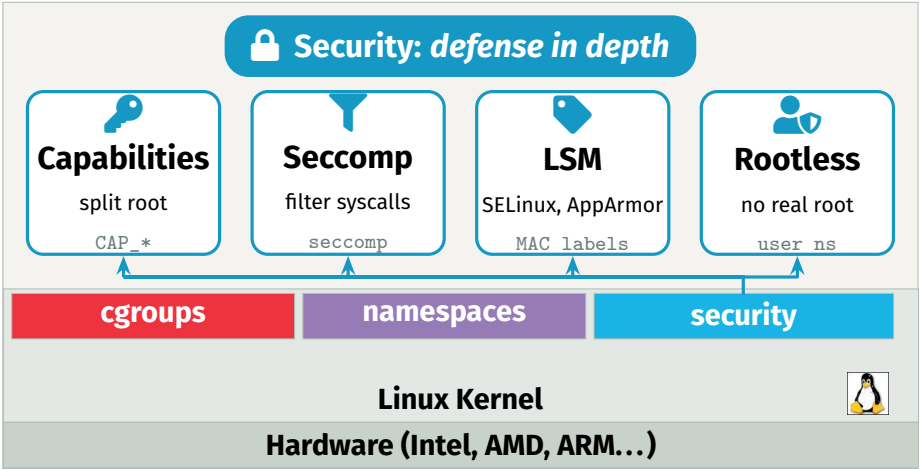
Why "Killed"? `tail` keeps the last *lines*; `/dev/zero` has no newline → one endless line → `memory.max`, no swap: nothing to reclaim

- The OOM killer, *scoped to this cgroup*, `SIGKILL`s its biggest process: `sh` survives, the host does not notice
- Killed by signal $N \Rightarrow$ exit status $128 + N$: $128 + 9 = 137$ (143: `SIGTERM`)
- Main process killed: `Exited (137)`, `OOMKilled: true`

No limit: the *global* OOM killer may pick any host process. Engine flags → cgroup files: appendix

Lab: Module 1, Ex. 1.3

Linux Container Architecture: Security



also: `no_new_privs`, read-only root filesystem, cgroup limits

Containers Do Not Contain

Containers do NOT contain!!!

A container is a process on a **shared kernel**.
One kernel bug, one misconfiguration, or one dangerous flag and the isolation is gone.

- Not everything is namespaced: kernel modules, keyrings, parts of `/proc` and `/sys`, the clock, devices…
- One shared kernel = one shared attack surface → engines stack several layers (defense in depth)


RENA
MONROVIA

Three Locks on Every Container

	What it does	Key flags
Capabilities	root's power split into ≈ 40 flags; Podman keeps 11 (not `CAP_SYS_ADMIN`)	`--cap-drop=all`, then `--cap-add` what is needed
Seccomp	BPF filter on system calls: risky ones fail (`kexec_load...`); `no_new_privs`: setuid gains nothing	`--security-opt` + `seccomp=prof.json` or `no-new-privileges`
LSM	mandatory access control, even for root: SELinux labels (one per container), AppArmor	`:z` / `:Z` on bind mounts (lecture 2)

```
$ podman run --rm --cap-drop=all --security-opt=no-new-privileges \
      docker.io/library/alpine:3.20 chown nobody /tmp
chown: /tmp: Operation not permitted    root, but without CAP_CHOWN
```

⚠ On by default. Never `--privileged` (drops all three), never disable SELinux

Rootless, and What Breaks Isolation

Rootless

- Engine and containers run as *your* user, no root daemon
- Container root = your UID: an escape lands on an unprivileged account
- Default: Podman rootless, Docker rootful
- Costs: no ports < 1024 (default), user-space networking (pasta), some devices and sysctls unavailable

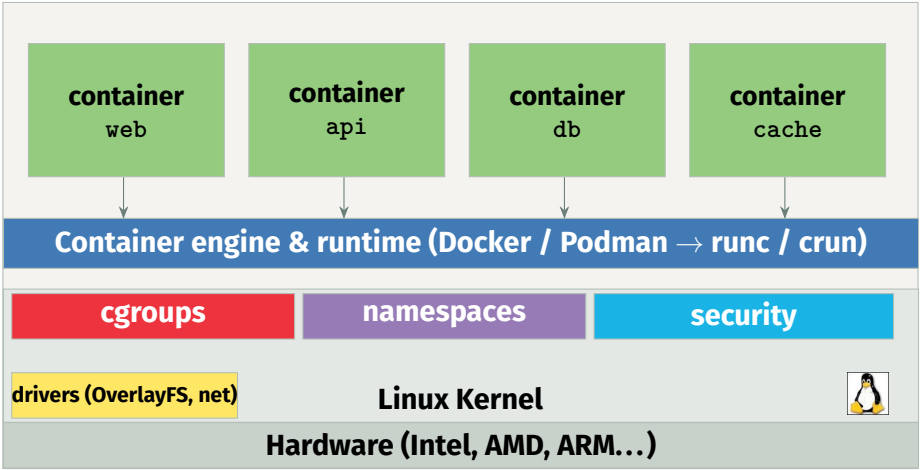
What breaks isolation

- Docker socket access = root on the host:

```
docker run --privileged \
  -v /:/hostfs -ti \
  docker.io/library/alpine:3.20
```

- `--privileged`, `-v /:/host`, `--device`, `--network=host`, `--pid=host`
- Kernel vulnerabilities
- Copy-pasted scripts and Compose files!

Linux Container Architecture: Drivers and Engine



OverlayFS: Union Mount + Copy-on-Write

- **Copy-on-write**: looks like a copy, is a reference; real copy only on *modification* (`fork()`, snapshots, VMs, containers)
- **Union mount**: several directories seen as one: read-only image layers + one writable container layer
- OverlayFS (Linux 3.18+): the default everywhere (AUFS is history; Btrfs, ZFS: snapshot drivers, not union filesystems)

`lowerdir`	image layers, read-only; `:`-separated, leftmost on top
`upperdir`	container layer, read-write: all changes land here
`workdir`	empty scratch directory, same filesystem as `upperdir`
`merged`	the unified view: the container's `/`

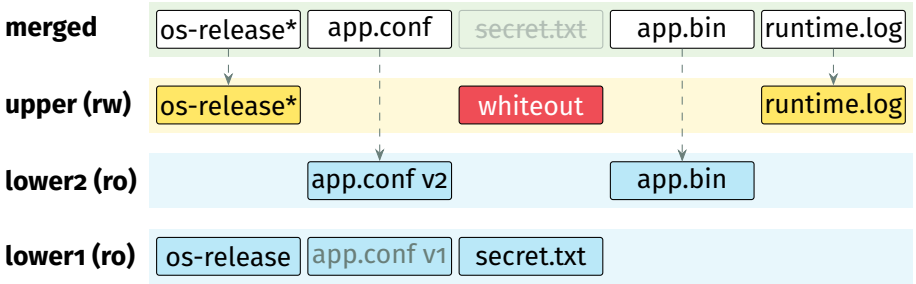
Lab: Reads, Copy-Up and Whiteouts

In the Ex. 1.1 sandbox, `lower1` = base OS, `lower2` = app layer (setup, steps 1–3: appendix):

```
# mount -t overlay overlay -o lowerdir=./lower2:./lower1,\
    upperdir=./upper,workdir=./work ./merged
# cat merged/app.conf
Config v2 (overridden by layer 2)          top layer wins
# ls -A upper                                empty
# echo "Modified at runtime" >> merged/os-release.txt
# rm merged/secret.txt
# echo "New runtime log" > merged/runtime.log
# cat lower1/os-release.txt
Ubuntu 24.04 Base Layer                    untouched
# ls -l upper
-rw-r--r-- 1 root root   57 ... os-release.txt   copy-up
-rw-r--r-- 1 root root   16 ... runtime.log      new file
c--------- 2 root root 0, 0 ... secret.txt       whiteout
```

Lab: Module 1, Ex. 1.2 (steps 4–6)

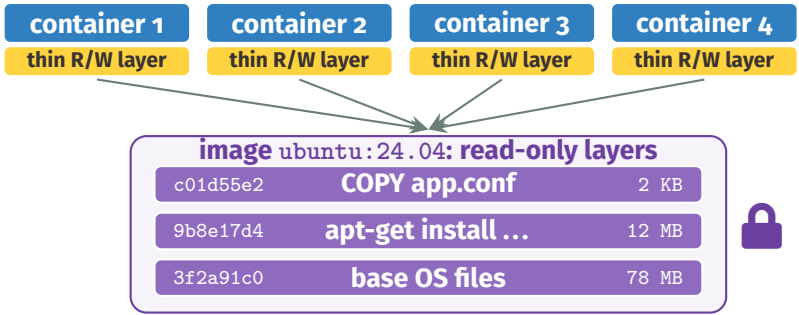
How OverlayFS Resolves Files



- **Read**: the top-most layer that has the file wins (reads fall through)
- At start, `upperdir` is empty: a new container costs nothing
- **Write**: copy-up of the *whole* file to `upperdir`, then modify
- **Delete**: a whiteout (char device 0,0) hides the lower file

Lab: Module 1, Ex. 1.2

Containers Share the Image Layers



per container: its changes only (`upperdir`), empty at start

shared: once on disk, once in RAM cache (`lowerdir`)

⚠ First write copies the *whole* file → write-heavy data in volumes
⚠ Deleted in a later layer ≠ gone: still shipped (secrets!)

Lab: Module 1, Ex. 1.2; Module 4

Putting It Together: What `podman run` Does

Podman and the OCI runtime (`crun`, `runc`): only **ordinary system calls**

1. **Podman** mounts the image layers + an empty layer	→ OverlayFS
2. **crun** `clone(2)`s a process in new namespaces	→ Namespaces
3. writes its `uid_map` (`newuidmap`, `/etc/subuid`)	→ user ns
4. puts it in a new cgroup, writes `memory.max`…	→ cgroups
5. mounts `/proc`, `/sys`, `/dev`, volumes; `pivot_root(2)`	→ mnt ns
6. drops capabilities, loads seccomp, sets the SELinux label	→ Security
7. `execve(2)`s the entrypoint → PID 1	→ pid ns

`podman exec`: `setns(2)` joins the *existing* namespaces.

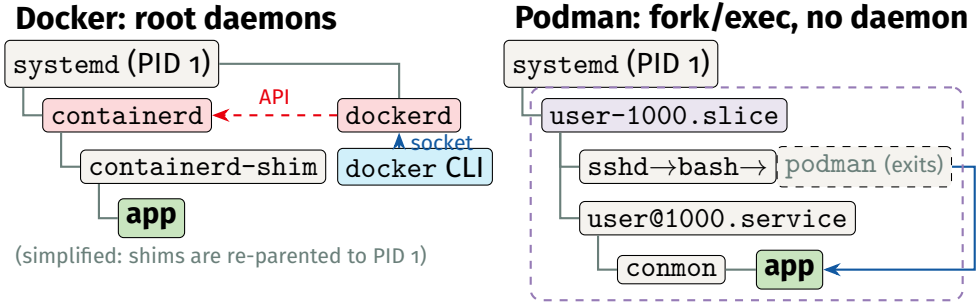
Lab: Module 1, introduction

Recap: Primitive, Flag, Where to Look

Primitive	Podman flag	Where to look
Namespaces: what it *sees*	`--hostname`, `--network`, `--pid`	`ls -l /proc/<pid>/ns/`, `lsns`
User ns: who it *is*	`--userns=keep-id`, `/etc/subuid`	`/proc/<pid>/uid_map`, `podman unshare`
cgroups v2: what it *uses*	`--memory`, `--cpus`, `--pids-limit`	`.../libpod-<id>.scope/memory.max`; exit 137, `OOMKilled: true`
Capabilities, seccomp: what "root" *may do*	`--cap-drop=all`, `--security-opt`	`grep -E 'Cap|Seccomp' /proc/<pid>/status`
SELinux, AppArmor: what it may *touch*	`:Z` / `:z` on bind mounts	`ps -eZ | grep container_t`, `ls -Z`
OverlayFS: how its `/` is *stored*	`--read-only`; volumes bypass it	`podman inspect -f '{{.GraphDriver.Data.UpperDir}}'`

`<pid>` = the container's PID 1 on the host: `podman inspect -f '{{.State.Pid}}' web`

Who Keeps Your Containers Alive?



- Docker: root daemons supervise, and **restart containers at boot**
- Podman: `conmon` supervises inside your user slice. Last SSH logout ⇒ `logind` stops `user@1000.service` ⇒ even `-d` containers die; and at boot, nobody starts them

Lab: Module 5

User Lingering: `loginctl enable-linger`

```
loginctl show-user $(whoami) -p Linger     # Linger=no
loginctl enable-linger $(whoami)
ls /var/lib/systemd/linger/                # marker file: <username>
# Compose stacks: start their "restart: always" containers at boot
systemctl --user enable podman-restart.service
```

- `user@1000.service` now **starts at boot** and **survives logout**: containers keep running (undo: `disable-linger`)
- `podman-restart.service`: no ordering, no health, no journal per service ⇒ **Quadlets** (next)

⚠ Refused over SSH (polkit)? `sudo loginctl enable-linger <user>` or ask the admin

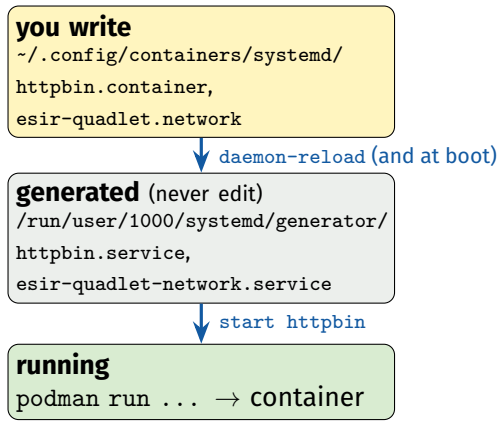
WSL2: enable systemd (`/etc/wsl.conf`: `[boot] systemd=true`).

Lab: Module 5, Ex. 5.1

Quadlets: Containers as systemd Units

- Podman $\geq$ 4.4: containers **declared** in systemd INI files
- Types: `.container` `.volume` `.network` `.image` `.build` `.kube` `.pod`
  pod = containers sharing one network namespace (appendix)
- systemd gives: boot start, ordering, restarts, cgroup limits, journal

~~`podman generate systemd`~~: deprecated



Lab: Module 5, Ex. 5.2–5.3

`httpbin.container` (1/2), Network and Volume Units

`httpbin.container`

```
[Unit]
Description=Hardened go-httpbin (Quadlet)
Wants=network-online.target     # (*)
After=network-online.target     # (*)
[Container]
Image=docker.io/mccutchen/go-httpbin:v2.15.0
ContainerName=quadlet-httpbin
Network=esir-quadlet.network
PublishPort=127.0.0.1:8080:8080  # loopback only
PidsLimit=128           # pids.max: stops fork bombs
ReadOnly=true           # read-only /: no webshell
NoNewPrivileges=true    # setuid gains nothing
AutoUpdate=registry     # podman auto-update
```

Refers to the **file** (`Network=`, `Volume=redis-quadlet.volume:/data`) ⇒ **dependency**, started first

Lab: Module 5, Ex. 5.2

`esir-quadlet.network`

```
[Network]
NetworkName=esir-quadlet-net
Subnet=10.89.50.0/24
```

`redis-quadlet.volume`

```
[Volume]
VolumeName=esir-quadlet-data
```

Same as `podman network|volume create`, **managed by systemd** (`NetworkName=`: name in Podman).

(*) not needed in user units: Quadlet adds its own network wait

`httpbin.container` (2/2): `[Service]` and `[Install]`

```
[Service]
Restart=always
RestartSec=5s
MemoryMax=64M
CPUQuota=50%
[Install]
WantedBy=default.target
```

- `[Service]`: copied into the generated unit
- `Restart=always`: restart whatever the exit status, after `RestartSec`
- `MemoryMax`, `CPUQuota` ⇒ cgroup v2 `memory.max`, `cpu.max`

`[Install] WantedBy=default.target`

- replaces `systemctl enable`: start with the user manager (at boot, with linger)
- generated units cannot be enabled: `systemctl enable` fails on them

Lab: Module 5, Ex. 5.2

Compile, Start, Audit, Crash

```
/usr/libexec/podman/quadlet -dryrun -user   # check the units
systemctl --user daemon-reload              # run the generator
systemctl --user start httpbin.service      # network unit too
curl 127.0.0.1:8080/ip
journalctl --user -u httpbin.service -n 20 --no-pager
# Crash it: SIGKILL the container's PID 1
kill -9 $(podman inspect -f '{{.State.Pid}}' quadlet-httpbin)
sleep 6; systemctl --user status httpbin.service
```

Crash recovery: PID 1 killed ⇒ exit 137 (128 + 9) ⇒ unit fails ⇒ `Restart=always`: a **fresh** container after `RestartSec=5s` (`ReadOnly`: nothing lost)

Debian/Ubuntu: `/usr/lib/podman/quadlet`. Podman ≥ 5.6: `podman quadlet install|list|print|rm`.

Lab: Module 5, Ex. 5.3

From `compose.yaml` to Quadlets

`compose.yaml`	**Quadlet** (`[Container]` unless noted)	**Note**
`image:`	`Image=`	pinned FQIN
`ports:`	`PublishPort=127.0.0.1:8080:8080`	front-end only
`volumes:`	`Volume=pgdata.volume:` `/var/lib/postgresql/data`	+ a `pgdata.volume` file
`environment:`	`Environment=`, `EnvironmentFile=`, `Secret=`	`Secret=`: `podman secret`
`healthcheck:`	`HealthCmd=` + `Notify=healthy`	"started" = healthy
`depends_on:`	`[Unit] Requires=postgres.service` `After=postgres.service`	≡ `service_healthy`
`networks:`	`Network=backend.network`	+ a `backend.network` file
`restart:`, `mem_limit:`	`[Service] Restart=always`, `MemoryMax=`	applied by systemd

`podlet compose` drafts the units from a `compose.yaml` (review them); full capstone units: **the field guide**

Lab: Module 4 stack, Module 5

Compose or Quadlets?

	`podman compose`	**Quadlets**
Description	one `compose.yaml`	one file per object
Supervised by	`conmon`, in your user slice	**systemd**, one unit each
After logout	killed, unless linger	runs (linger)
At boot	`podman-restart.service`	`WantedBy=default.target`
Restarts	`restart:`	`Restart=`
Limits	`mem_limit:`	`MemoryMax=...`
Logs	`compose logs`	`journalctl`
Updates	manual `pull` + `up`	`podman auto-update`
Best for	dev, quick stacks	**single-host prod.**

Same concepts, same kernel features: only the *supervisor* changes. For production, Quadlets are the **robust** option.

Lecture 3: What to Remember

- **No "container" in the kernel**: namespaces (what it *sees*), cgroups (what it *uses*), security (what it *may do*), OverlayFS (its files)
- **User namespace**: container root = your UID; other UIDs from `/etc/subuid`
- **cgroups v2**: over `memory.max` → OOM kill → exit 137 (128 + SIGKILL)
- **Containers do not contain**: shared kernel; drop capabilities, keep seccomp + SELinux, never `--privileged`
- **Production**: linger: containers survive logout; Quadlets = systemd units (restart, boot, journal)

→ Next: the course conclusion lectures 1–3

what to remember from the whole course; what was not covered; going further

Not Covered Here

Supply chain

- signing: cosign, sigstore
- SBOM: `syft`, `build --sbom`
- scanning: Trivy, Grype
- multi-arch images: `--platform`, manifest lists

Other worlds

- dev containers, Toolbx
- Kubernetes, multi-host networking (details: appendix)
- Windows containers
- WebAssembly (Wasm)

Same building blocks: OCI images, registries, namespaces, cgroups.

The Whole Course: What to Remember

M0 Environment	a process on the host's Linux kernel (Windows/macOS: a Linux VM); runs on *any Linux kernel of the same CPU architecture*
M1 Kernel	namespaces: what it sees · cgroups: what it uses · OverlayFS: its layers · exit 137 = OOM kill ($128 + 9$)
M2 Images	immutable OCI layers; full names, pinned tags or digests; a container lives as long as its PID 1
M3 Storage, net	rootless = UID mapping (`:Z`, `:U`); named volumes; DNS networks; publish only the front, on `127.0.0.1`
M4 Build, Compose	cache-friendly order, multi-stage, non-root `USER`, no secrets in layers; health checks + `service_healthy`; segmented networks
M5 Production	systemd supervises: linger + Quadlets (`Restart=`, limits, journal, auto-update)

Trade-off: lighter and faster than VMs, but a shared kernel = weaker isolation.

Going Further

Online

Container training (J. Petazzoni et al.)	`https://container.training/`
Podman (rootless, Quadlets, man pages)	`https://docs.podman.io/`
Docker	`https://docs.docker.com/`
OCI image, runtime, distribution specs	`https://opencontainers.org/`
Kubernetes	`https://kubernetes.io/docs/`

Books

Liz Rice, *Container Security*, O'Reilly, 2020
Daniel J. Walsh, *Podman in Action*, Manning, 2023

Man pages

`namespaces(7)`, `user_namespaces(7)`, `cgroups(7)`, `unshare(1)`, `podman-systemd.unit(5)`

This course: the appendix (deep dives) and the field guide (troubleshooting, HTTPS with Caddy, backups, the capstone as Quadlets)