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

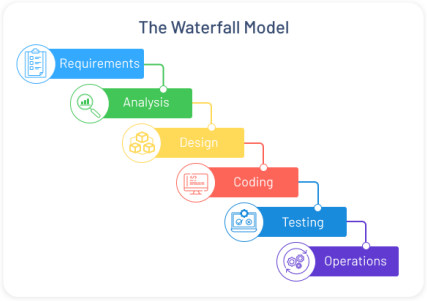
1. **DevOps** — from waterfall, CI/CD, tools
2. **Microservices** — cloud native, scaling
3. **Why Containers** — dependencies, shipping analogy
4. **Containers vs VMs** — shared kernel, trade-offs
5. **Image vs Container** — layers, vocabulary

lab Module 0

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

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

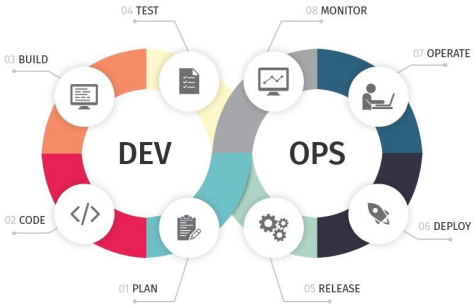
The Waterfall Model



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

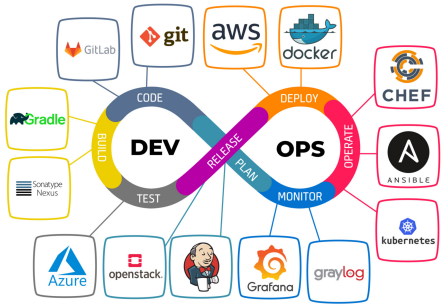


DevOps = Development + Operations



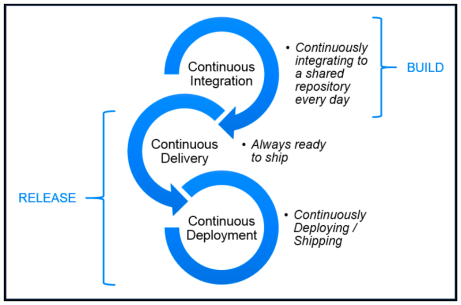
One team owns the whole loop, from plan and code to deploy and monitor.

Popular Tools



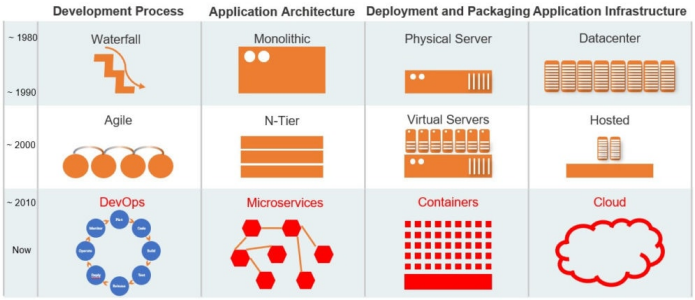
Every stage is automated; containers carry the app from build to run.

Continuous Integration, Delivery, Deployment

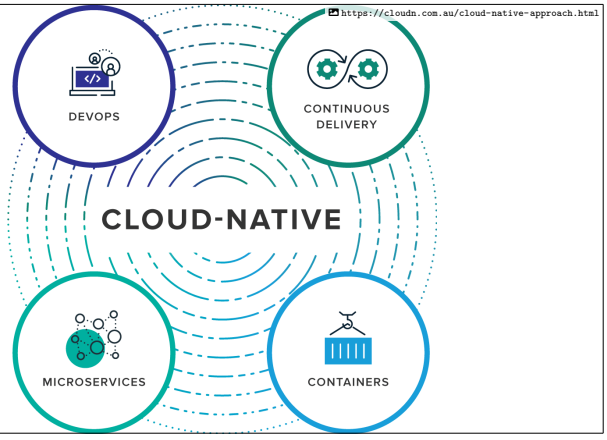


Integrate daily, always ready to ship: the same environment at every step.

Software Architecture Over Time



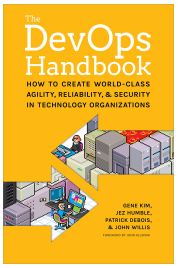
Monolith on physical servers → microservices in containers, in the cloud.



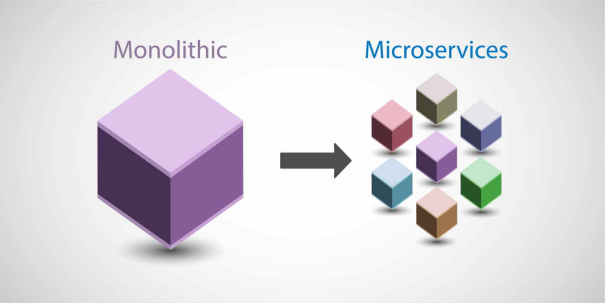
Reading

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

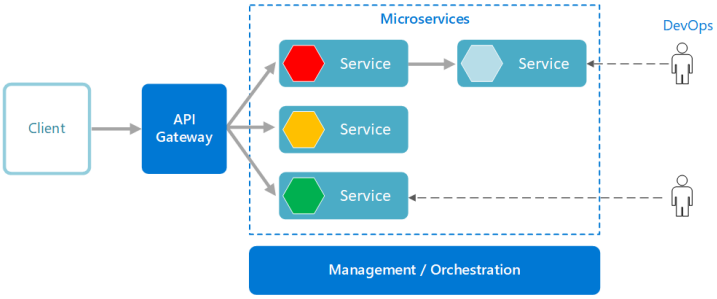


Towards Microservices



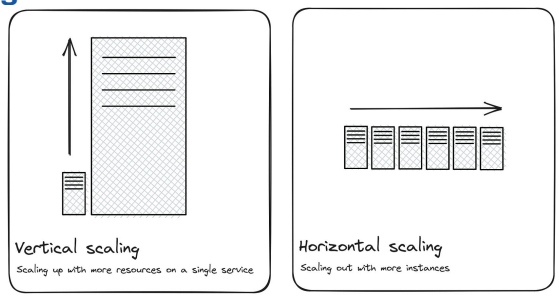
Split the monolith into small services, each deployed and scaled on its own.

Microservice Architecture



Dozens of services × environments × daily deploys…
… we need one standard, self-contained unit: the **container**

Scaling: Vertical vs Horizontal



Containers make horizontal scaling cheap: start more identical instances.

Docker, Podman & Friends

- **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)

`https://www.docker.com/` `https://podman.io/`

Why Containers?

Problem

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

```
mysensors-proxy:
nginx => 80
site: 6th.io
proxy: forum.mysensors.org => localhost:4567
proxy: mysensors.org => localhost:3000
```

```
mysensors-docs:
meteor (nodejs) => 3000
mongodb => 27017
```

```
mysensors-forum:
nodebb (nodejs) => 4567
redis => 6379
```

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

×

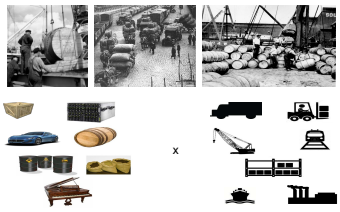
Deployment environments

- 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 × 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

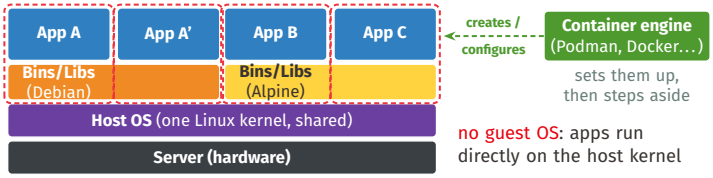
Developer concerns (inside)
- code, libraries, services
- configuration, data
- *all servers look the same*

Ops concerns (outside)
- moving, starting/stopping containers
- logging, monitoring, network config.
- *all containers look the same*

N apps × M environments ⇒ $N + M$
package each app once, teach each environment to run containers once

What Is a Container?

one container = a process + its files, in *namespaces* + *cgroups*



no guest OS: apps run directly on the host kernel

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

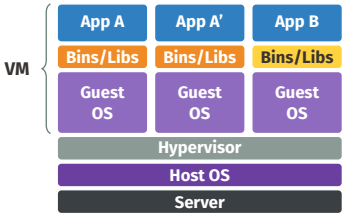
- Kubernetes, on any cloud or on-premises
- "serverless" (Cloud Run, Fargate)
- CI jobs, a fresh container each
- your own VM + Podman (assignment!)

What you use them for

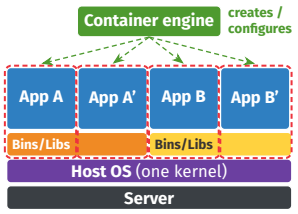
- any service version, one command
- test on several distros, no VMs
- reproducible dev environments, CI
- integration tests (Testcontainers)

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

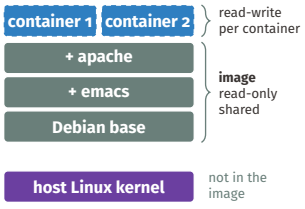


Image	Container
program on disk	→ process
class	→ object (instance)
read-only layers	→ + one writable layer
+ metadata (cmd, ports…)	
built once, shipped	→ run many times

```
>_ 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