CONTAINERS

Yohai Meiron

2026 October 6

AGENDA

- What containers are
- The underlying technologies
- Containers in practice
  - Container engines
  - Usage examples
  - Advanced topics

We'll be discussing the *Linux* operating system only

WHAT IS A CONTAINER?

My very broad definition:

> "A container is a *sandboxed* process (or group of processes)
> that is (are) running from an *image*
> that contains most or all of the dependencies"

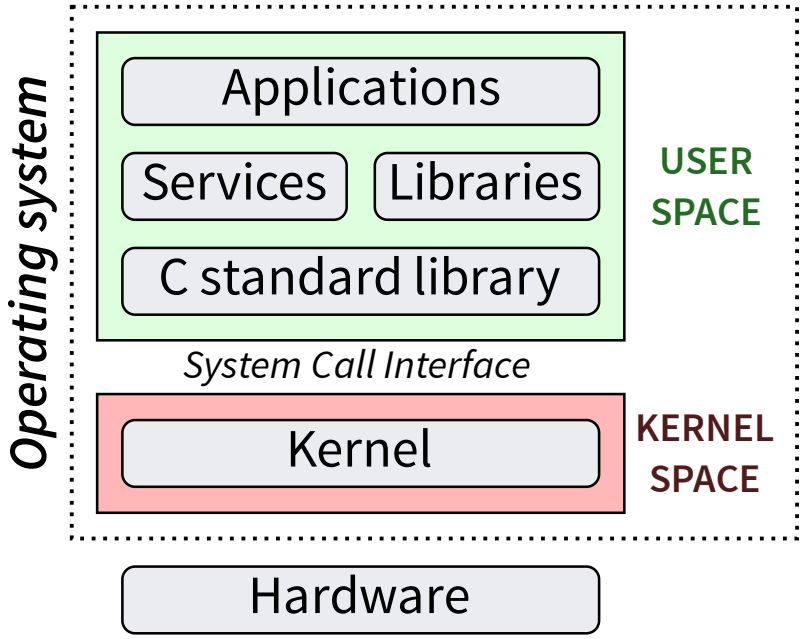
WHAT IS IT FOR?

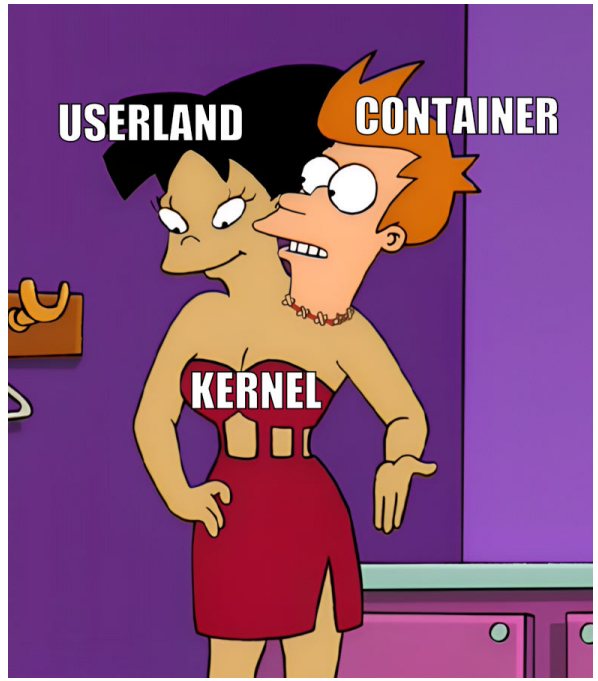
(Especially in the HPC context)

Ease:	Fetching an image is often much easier than "bare metal" installation
Reproducibility:	Shipping a program with all its dependencies helps reproducibility
Control:	It's possible to control access to resources

KERNEL SPACE AND USER SPACE

- The Linux kernel and “userland” processes run in separate memory spaces





- Humans interact with userland programs, hardware interact with the kernel
- Userland doesn’t care what exact kernel is running (and vice versa)
  - It just sees the system call interface
- The modern Linux kernel can support multiple userlands *simultaneously*
- That’s what a container is (alternative definition):

> “A container is a (usually minimal) userland that straps on to a running OS”

CONTAINER IMAGES

An *image* is the collection of files that are needed to run the process(es)

- Files are organized per the *Filesystem Hierarchy Standard* (`/bin`, `/etc`, `/lib`, etc.)
- It includes empty directories to serve as mountpoints for pseudo file systems (`/dev`, `/proc`, `/run`, etc.)

How much space (in MB) do OS components take? (Ubuntu Server 24.04.2)

Category	Ubuntu Server image	Ubuntu Docker image
Kernel[+]	704	0
Libraries	472	50
Devel	261	0
Python	195	0
Admin	130	21
Utils	43	15
Other	235	11
Total	**2040**	**97**

A container image only has *userland* components, and usually just essential ones

CONTAINER IMAGES (CONT.)

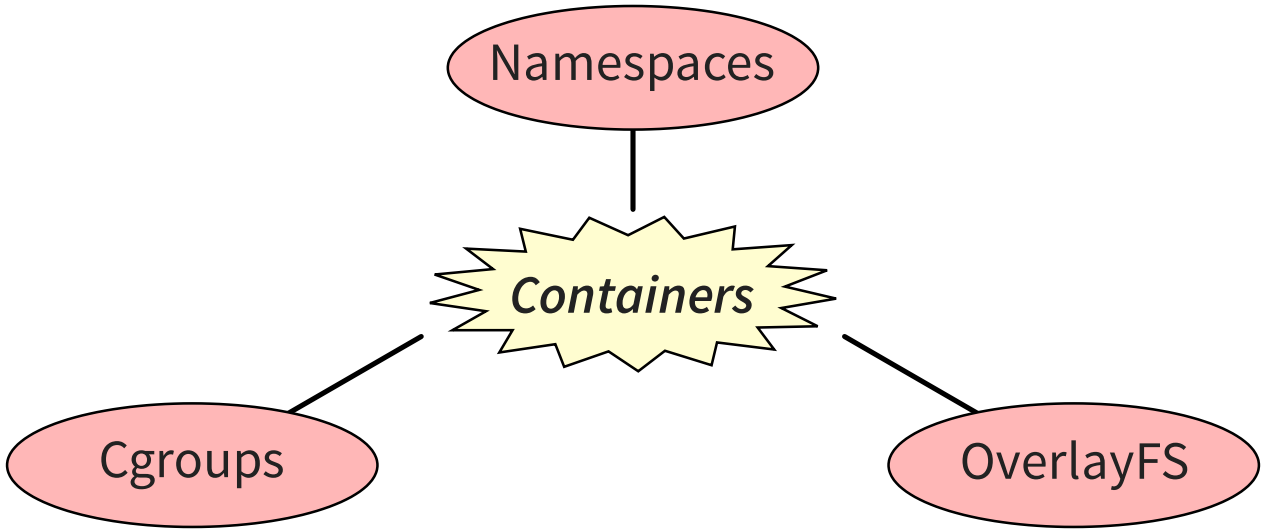
What's usually included in a *base* image?

- ELF Interpreter / dynamic linker (e.g. `ld-linux-x86-64.so.2`)
- Standard C library (e.g. glibc)
- Shell (e.g. Bash)
- Utilities (e.g. `ls`, `cp`, `mv`, etc.)
- A package manager (e.g. APT)
- System-wide configuration files (i.e. the `/etc` directory)
- Common libraries (e.g. libz, libstdc++, libpthread, etc.) as needed
- Tools (e.g. curl, git, etc.) as needed

Alpine Linux Docker image is only 8 MB!

⚠️ Container images are CPU architecture-specific (e.g. x86-64, ARM64, …) ⚠️

KERNEL FEATURES ENABLING SANDBOXING



These technologies are *hierarchical*: it's possible to run a container within a container (within a container etc.)

NAMESPACES

It's like userland "parallel universes"…

Used to isolate ("sandbox") certain aspects of container processes from the host and from each other

- Mount
- Process ID
- Network
- User ID
- Others…

Can be interfaced via syscalls (e.g. `unshare` and command line utility of the same name)

CONTROL GROUPS (CGROUPS)

A group of processes that is controlled or monitored by a kernel component (resource controller)

- cpu, cpuset
- memory
- io
- pids
- Others…

Also used in job schedulers in HPC (e.g. Slurm) to limit resources available for job

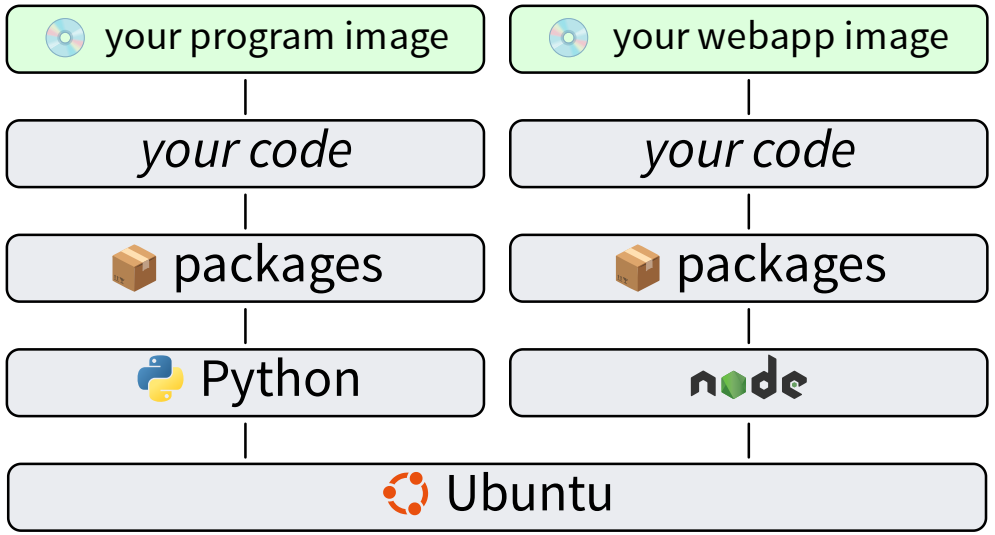
Can be interfaced via:

- pseudo file system (easy)
- BPF programs (hard)

OVERLAYFS

It's a virtual view of a file system that combines *multiple directories* into one (union mount)

- An image is often made up of multiple overlaid layers



- OverlayFS can make a read-only file system (virtually) writable

Can be interfaced via the `mount` syscall and command line utility of the same name

CONTAINER ENGINES

Putting it all together…

It is a software tool that usually has these abilities:

- download (“pull”) images from the internet
- run containers of off images
- set up resources (storage, networking, CPUs…) for containers
- interact with running containers (attach a shell)
- manage running containers (status check, stop)
- build images and upload (“push”) to the internet

CONTAINER ENGINES (CONT.)

Modern Linux container engines include:

	engines	category
1	docker podman	services
2	Apptainer	HPC applications
3	LXC LXD	virtual machine-like
4	Flatpak Snap	desktop applications

All use the same underlying core technologies and are *somewhat compatible*

So which one should I use??? Doesn't matter. Also, Apptainer probably…

APPTAINER

Formerly know as “Singularity”

- Very easy to get started with
- Convenient integration with the host OS
- The `.sif` image format is portable
  - A single self-contained file that’s easy to transfer
  - Use it where you would normally use an executable (interactive or batch)

APPTAINER DEMO

We have a data file and a Python script that processes it in some way and outputs the result to screen.

The script depends on **Pandas** (and its dependencies), a Python package that is not installed on our system.

We can install them using our distro's package manager, or a third party package manager such as Mamba.

But this installation process will be different on different systems. The more dependencies there are, the more difficult it is to reproduce the environment across *different systems* and across *time*.

So let's use Apptainer to build an image with all the dependencies.

Later we'll include the script itself in the image, creating a "containerized application".

COMMAND WE'LL USE:

```
apptainer pull --sandbox my-image/ docker://ubuntu
apptainer shell --writable --fakeroot my-image/
Inside container:  apt update && apt install python3 python3-pandas
apptainer exec my-image/ python3 analysis.py iris.csv
apptainer build my-image.sif my-image/
```

APPTAINER DEMO (CONT.)

What we did was cool and all but:

- Difficult to reproduce, especially if we made a lot of modifications to the base image
- The image only has dependencies, we'd need to distribute our application (`analysis.py`) separately

```
1  Bootstrap: docker
2  From: ubuntu:24.04
3  Stage: build
4
5  %setup
6      mkdir -p ${APPTAINER_ROOTFS}/opt/bin
7
8  %files
9      analysis.py /opt/bin
10
11 %post
12     export DEBIAN_FRONTEND=noninteractive
13     apt update
14     apt install -y python3 python3-pandas
15     apt clean
16
17 %runscript
18     exec python3 /opt/bin/analysis.py "$@"
```

Build it with:

```
apptainer build my-image.sif my-image.def
```

Run with

```
apptainer run my-image.sif iris.csv
```

(run ≠ exec)

or just

```
./my-image.sif iris.csv
```

What if the data file is in some special path like `/scratch/user`?
Apptainer only *bind mounts* `$HOME` by default (as well as pseudo file systems), but not arbitrary paths

Just bind the directory manually:

```
apptainer run --bind /scratch/user my-image.sif /scratch/user/iris.csv
```

DOCKER

- More popular than Apptainer (by far)
- Used for services (webapps, database), software development pipelines…
- Not very suitable for multi-user systems (cf. Rootless mode)
- Cloud-friendly (targeted by an ecosystem of tools)
- Prefers isolation over integration
  - Does not mount `$HOME` by default
  - Container network isolated by default (i.e. ports)

```
1 FROM ubuntu:24.04
2 ARG DEBIAN_FRONTEND=noninteractive
3 RUN apt update \
4     && apt install -y python3 python3-pandas \
5     && apt clean
6 RUN mkdir -p /opt/bin
7 COPY analysis.py /opt/bin
8 ENTRYPOINT ["python3", "/opt/bin/analysis.py"]
```

Here is the equivalent *Dockerfile*

Build it directly with:

```
docker build --tag my-image .
```

Run with

```
docker run --volume /home:/home my-image /home/user/iris.csv
```

COMMENTARY ABOUT THE DEMOS

- For both Apptainer and Docker, these just scratch the surface
  - Check out the respective documentations
- Ubuntu isn't such a great base image
  - Use Alpine in most cases
- It's not such a great idea to use the system's package manager (APT) to manage a Python environment
  - Use a virtual environment or Micromamba
  - Prepare a `requirements.txt` file with exact versions
- Each `RUN` in the Dockerfile creates another layer
  - Combine operations with `&&` and break lines with `\`
- Take advantage of layer caching by putting frequently changing steps near the end of the Dockerfile
- Podman is a very good Docker alternative