Leibniz Supercomputing Centre
of the Bavarian Academy of Sciences and Humanities

Introduction to Containers & Application to AI at LRZ

Theory & Practice

Florent Dufour · florent@lrz.de

Big Data & Artificial Intelligence
Leibniz Supercomputing Centre
October 23, 2023

Module description

Since the introduction of Docker back in 2013, software containers have become the industry standard for software packaging, distribution, and deployment.

Creating a container consists in bundling an application, its dependencies and runtime in a single unit that can later run independently of the underlying infrastructure. Unlike virtual machines, containers are lightweight and yield higher performances while providing greater versatility and interoperability. As containers accommodate an easy, safe, reliable, and scalable way to run applications and pipelines, they are an attractive candidate for High Performance Computing (HPC) and Artificial Intelligence (AI) workloads.

With this module, we will showcase the most enticing features and niceties offered by containers. Not only will we explore their history and implementations, but we will also dive into actual and advanced uses with a particular emphasis on Artificial Intelligence tasks, reproducible biomedical pipelines, and automated workflows. Participants will roll up their sleeves and get their hands on the LRZ Compute Cloud to set containers in action. By the end of the course, participants will be able to transfer their knowledge and experience to their specific use-cases and requirements.

The content of this document is organized as follow: In section 1, we will explain where containers come from, what problems they solve, and how do they compare to other solutions like Virtual Machine (VM)s. The first hands-on will help us familiarize with Docker containers. We will experiment and see that they are volatile and disposable. Section 2 will be the opportunity for us to see how one can build a custom image, and map specific volumes and ports from the host machine to a container. We will practice the art of the Dockerfile during the second hands-on and make an Artificial Neural Network (ANN) dream in a container. After reaching the limits of Docker in terms of security and performances, we will review other solutions that are more suited to HPC, specifically enroot and Charliecloud. We will use our experience with Docker to build images and convert images. We will go one step further with section 3 and see that we can provide hardware acceleration to containers and easily deploy them to large scale systems. We will emphasize that in order to get the full benefit of containers, they must be abstracted, with example in reproducible software builds and scientific pipelines. Before concluding, we will run a complex RNA-Seq analysis in a reproducible manner by making Nextflow leverage containerized pipelines. References are provided all along, and appendices can be used at any time to provide help and directions.

This is document revision: 2023.2a
Latest revision available at: https://doku.lrz.de/x/eQBvB

Table of Contents

List of Figures

List of Tables

List of Code snippets

Glossary

AI Artificial Intelligence.

ANN Artificial Neural Network.

API Application Programming Interface.

CI/CD Continuous Integration / Continuous Deployement.

CLI Command Line Interface.

CPU Central Processing Unit.

DSL Domain-Specific Language.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HPC High Performance Computing.

LRZ Leibniz Supercomputing Centre.

ML Machine Learning.

MPI Message Passing Interface.

NGC NVIDIA GPU Cloud.

OS Operating System.

RAM Random-Access Memory.

SLURM Simple Linux Utility for Resource Management.

UDSS User Defined Software Stack.

VM Virtual Machine.

1 A Tour of Containers

By then end of this course, you will be able to define, build, and orchestrate containers. We will dive into all their ins-and-outs, review their history, explore the landscape of possibilities, and have our hands on actual code. You will quickly come to realize that containers are volatile and disposable software units and you should therefore have no scruple creating, hacking, and destroying them. There is nothing you can break. Buckle-up, off we go.

1.1 You Are Here

You may have already written some code in your life. Maybe for university group projects or for the company or institution you're working for. If not, let's get in the shoes of a web developer for a moment[1]. You're most certainly provided with a laptop and you write code locally. You maintain your working environment yourself with the set of software and frameworks you prefer. You install your favorite text editor, have multiple versions of Python, JavaScript, Java, PHP or Rust that you maintain with package managers, virtual environments, SDK managers, or tany tool of your choosing. Also, your code certainly relies on specific versions of libraries: some very common, others might be exotic. You may even have written your own librairies! At some point, your code will have to be delivered and serve its purpose in "production". From here, things will get wild. You might be the best developer in town, pushing your code to production is always a scary thing to do. Believe it or not, humans have even developed rituals and believes around that. For example, it is commonly accepted that you will never push code to production on a Friday evening[2]. People also never push to production when it is full moon, when there is less than 3 cups of coffee left in the machine, or when the outdoor atmospheric pressure drops below 1011 hPa. Pushing to production is indeed frightening because so many things that are not under your control can – and most likely will – go wrong. Technically, the shiny code you've spent hours writing with the most cutting edge and fancy technologies on the custom environment that is your laptop will run on some kind of mutualised server, within a Virtual Machine (VM), administrated by someone other than you: the operations team (call them "ops" when you're intimate enough). Not only this team are no developers, but they have pretty strict standards, stacks, and pre-baked environments with pinned software versions they support on their servers. But you shouldn't worry too much, in any case you're smart right? You've followed the instructions and pushed your application to the fresh production environment they prepared for you. You've updated all the config files and sourced all the requirements. It's time to start the application. Let's hit run and... the thing blows up. Your application is bleeding error logs, spitting alerts you're not sure you've seen before (including system errors(!)). Its companion database is as much in disarray and your boss is on the phone trying to remain calm while passively-aggressively asking you to get the application running, now. For you there is not much to say, unless maybe the infamous "Well, but it works on my machine". You SSH into the remote server, keep rebooting desperately wishing it will finally run as intended. Ops on the other line are bothered by your calls and can't be of great help because *you* are the developer of this thing, and *you* must be able to fix it.

We are exaggerating you say[3]? Wait to see what happens when you go through this process but with python scripts, Machine Learning (ML) libraries, TensorFlow, dependencies you have

[1] Although you may prefer to get into flip-flops if you work at Google, preferably with socks if you're a proud German.

[2] If you're unsure, you can refer to: estcequonmetenprodaujourdhui.info

[3] Give a quick look here: reddit.com/r/devhumor

to download and/or build from source yourself! This whole process is called *shipping*. In the above example, we ship code from your laptop (machine A) to the production server (machine B). Most likely, you will ship from your laptop to the test server, then pre-prod, then production. Shipping also happens between servers, when you contribute to an open source project, when you share your code with a colleague, when you migrate your application from an old production server to a cloud provider, or even when you scale your application. *Shipping code happens all the time*. But we're not done here, even when you successfully have your application running on machine B, you're not even completely done. As a matter of fact, you still have to make sure that your code behaves the same. *Shipping is the sinews of war.*

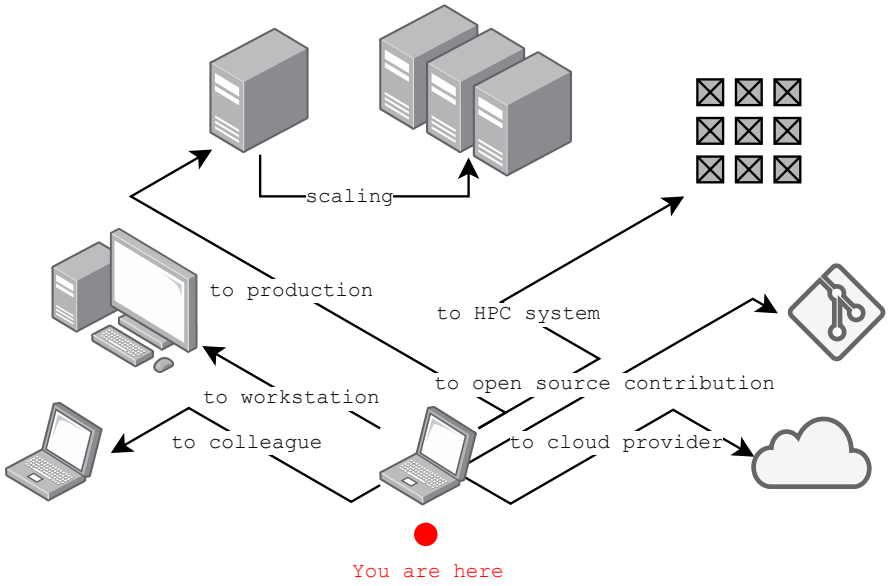

Figure 1: Shipping is the sinews of war

But now, let's imagine a beautiful world, somewhere sunny where if your code runs on your laptop, it will run exactly the same in production. Not only you know it will run smoothly when you push it, but you an also be assured that it will behave as intended. Even better, bugs you encounter locally should emerge in production as well: this is *reproducibility*. How can we achieve this? Well, in the end, if it works on your machine, why not simply ship your machine [1, 2]?

We will see that with containers you can ship your machine in a very lightweight and efficient manner. You can bundle your code, its runtime, dependencies, libraries, settings, config files etc. within a single software unit. With containers in production, you can be assured that your code will run as it should. It is such a powerful and efficient concept, that containers are now being repurposed for local and collaborative development [3, 4]. Not only do they bridge the gap between developers and ops, but they also solve issues in the High Performance Computing (HPC) world we've been facing for years, enabling researchers to take full advantage of massive compute resources in a secure, efficient, and flexible manner. This course is not trying to cover the full extent of containerization and dev/ops, but will remain focused on scientific applications, HPC, and ML/AI leveraging containers.

1.2 Basic Concepts for Containers

While this module is simply entitled "Introduction to Containers & Application to AI at LRZ", it is important to understand that this magic thing we call a container is only one of the 3 main layers of a larger software stack. This stack is composed of: containers, images, and orchestration tools. For now it might be a bit fuzzy but it's totally fine. Let's start from the beginning and give some definitions [2, 3]:

- **Image:** The way container software is shipped. An image is a static, standardized, and portable filesystem snapshot with a predefined executable command or entrypoint. Images are built and can be stored and distributed. You can think of an image as an executable that bundles all the dependencies making it very portable. Images are a tool for reproducibility used to ship fully built and packaged versions of application components, assets etc.
- **Container:** The way processes are run in a isolated environment. A container is a running instance of an image. A container runs on a host machine and shares the kernel with the Operating System (OS). The processes running in the container are isolated from the rest of the system. You can think of containers as lightweight ephemeral virtual machines (although we'll see they are fundamentally different in section 1.3). Containers are a tool for isolation where components are carefully segregated, providing increased security and resiliency.
- **Orchestrator:** What makes containers manageable. The orchestrator automates the lifecycle of containers. It organizes containers into abstract services and handles dependencies. It allows to declaratively describe how containers should behave. An orchestrator is a tool for managing complex applications by providing load balancing, monitoring, automated restarts, version migration, and many other convenience capabilities.

Alright! That's a lot. The point is not really to remember these definitions, but to know that they are here and come back to them later. Things will become clear once we will have our hands on a terminal and set containers in action! The most important thing for now is to understand that for the world to be a better place, we need to enable users to bring their stack to the production environment as opposed to having to deal with a pre-baked environment that may not fulfil their needs. This is usually referred to as User Defined Software Stack (UDSS): the user defines the stack in which the code will live in production, wether it is a server or a very complex and stringent HPC system. Containers allow UDSS to materialize in the real world.

1.3 Containers vs. the World

At this point, you may start to wonder why we bother talking about containers. It seems like they try to solve a very old problem that a bunch of solutions in the wild already claim to solve. Software packages, virtual environments, executable archives, sandboxed environments, or even Virtual Machine (VM)s are tools for packaging and shipping that we already know how to use.

As a Java developer for example, you may argue that you don't need containers: your code and the required dependencies are compiled and packaged into a static JAR file that can be easily be shipped and run. As a Python developer, you can take advantage of virtual environments to provide your application with an isolated view of its dependencies, or you can simply compile all your code and dependencies down to a big static binary that can be shipped and run directly. However, if you use these tool enough, you will reach their limitations. With both instances, you only sandbox your environment, and remain reliant on external components like

system libraries for example. Think about it, has any Java developer ever successfully shipped a JAR in production on the first try anyway? How many Python developers have < 36 virtual environments scattered on their drive, most of which are lost somewhere?

The best contender to containers really are Virtual Machine (VM)s. The idea goes as follow: If a sandbox of my environment is not enough, let's simply clone my entire laptop a let it run as a VM in production then! That sounds fair, a VM is a compute resource that uses software instead of a physical computer to run programs and deploy apps. One or more virtual "guest" machines run on a physical "host" machine thanks to an hypervisor. Each virtual machine runs its own operating system and functions separately from the other VMs, even when they are all running on the same host[5]. That's also isolation! Better, this might be the only reliable way to ship software: your development environment is in the end an integral part of your application. However, running your application in a VM is not really "Running your application in a VM" but much more: creating, booting, maintaining[4], updating and managing the VM that runs your application. This is far away from what you want/should be focusing on as a developer. You're not "ops"! Using VMs is in the end too heavy and equates with a large overhead. It is not practical for software delivery, especially not for intensive tasks like the one you may want to run on HPC or AI systems. Not to mention that VMs are usually several GB large and are not easy to ship or deploy. We show on tables 1 and 2 the advantages and disadvantages of VMs and containers [6].

On the other end, when it comes to containers, the process boils down to creating an image for your application and spin a container from it. "It just works"®. *Containers strike the right balance between the needs of performance, access to hardware, and freedom for the user to define the environment they want their application to exist in.* More about performances in section 2.4. Containers allow: i) to sandbox the entire system, ii) without machine details, and iii) without performance hit. Unlike a VM, a container does not emulate a machine, it runs on it. These differences are depicted on figure 2.

We are coming close to the first hands-on, let's review what we will need. The container technology stack (*i.e.,* the components installed on your machine) is composed of:

- **An image management system** that will allow you to build store, ship, and pull images.
- **A container runtime** that will run a container from the image.
- **A container management engine** that will manage the container.
- **An API** to access the engine that will be the bridge between you and the API.
- **A friendly mechanism to access that API** This can be a Command Line Interface (CLI) or Graphical User Interface (GUI) client for that API.

This is of course of lot. Lucky for us, we don't have much to worry about, most of this work has been taken care of and we can simply use a all-in-one solution. As you may already know, Docker is the standard solution in the industry, and for good reasons: it is very powerful while remaining approachable. Docker is the right place to start our container journey because as you will see, other UDSS like enroot or Charliecloud simply orbit around its ecosystem. At its core, Docker is a company (Docker Inc.) that has democratised container technology since the

[4]That implies: messing with systemd, emulating hard drives, turning services on/off, having to deal with network interfaces...

Table 1: Strengths and weaknesses of VMs

Strengths	
Flexibility	Freedom to run any kernel, Operating System (OS) and distribution with the set of software one wants. A VM behaves as an regular and full-fledged system.
Isolation	Strong to complete isolation: from the host, hardware, and other VMs.
Performances	Acceptable and predictable performances for common use cases. Besides, scaling a VM is usually easy.
Weaknesses	
Overhead	The infrastructure can be complex to setup; Entire OS to provision, boot, maintain (including at a low level: systemd, networking...)
Hardware access	Since software is used to emulate hardware, operating a VM implies trading simplicity over performances. Direct / performant access to hardware (Infiniband, Graphics Processing Unit (GPU)...) might not be possible or may require privileges
Security	While VMs provide strong isolation, they offer a larger attack surface compared to bare metal since many additional software components are required: hypervisor, cloud platform etc.
Not HPC friendly	A VM is appropriate for cloud applications, which is fundamentally different than HPC. Running VMs in a HPC environment is very uncommon because of the very precise set of constraints to be met in terms of performances, security, and implementation (see section 2.3)

Table 2: Strengths and weaknesses of containers

Strengths	
Lightweight	A container shares the kernel with the host. No complete system to provision. A container is lightweight in resources utilisation. More in section 2.1.
Isolation	Processes running in a container are strongly isolated to the host and other containers. More in section 2.1.
Performances	Bare metal performances (or close). More in section 2.4.
Simplicity	Extremely simple tooling and intuitive CLI.
Weaknesses	
Newness	Require recent linux kernel / distro (although it is less and less a problem). Still some rough edges; Not widespread adoption and expertise.
Security	Can be insecure if not set up appropriately and may require privileges to run. More in section 2.3.
Addictive	Putting stuff into containers can make your life so great that it has been reported to be highly addictive.

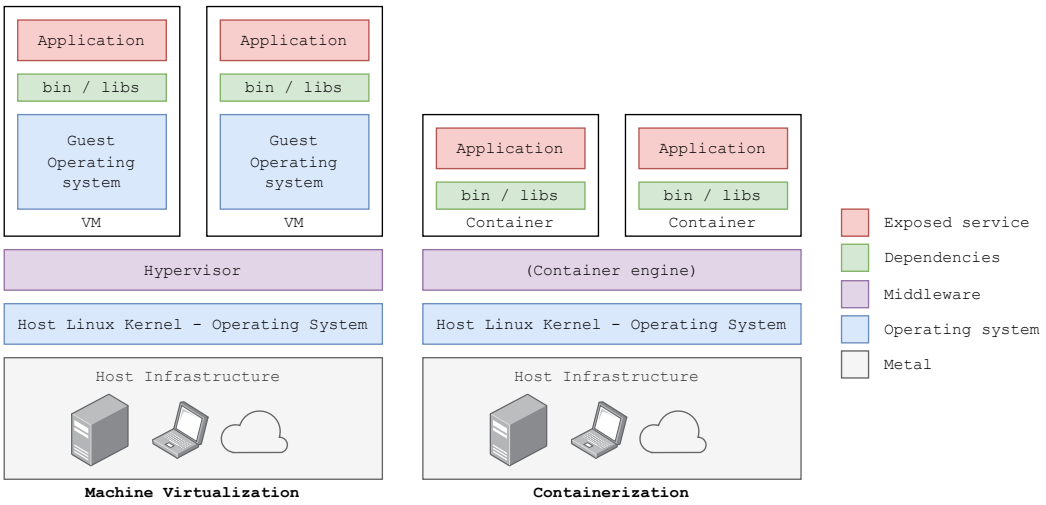

Figure 2: Comparison of the virtual machines (left) and containers (right) stacks

seminal talk of Solomon Hykes in 2013: "Why we built Docker", at the dotScale conference [2]. In addition to the stack we described before, Docker provides [7]: i) An open standard container format ii) A large amount of open-source, low-level container technologies, available for free[8]. iii) A way to build images and run containers locally (Docker desktop) with the Docker CLI. iv) A way to host/find/share images (Docker hub). It is some kind of app store for images v) A way to orchestrate container (Docker compose / Docker swarm)

1.4 Kickstarter: Your First Steps With Containers

Alright, that's enough for the theory! Let's get our hands on a terminal and have fun with containers. Let's SSH into your workbench in LRZ's compute cloud. Use the IP address, username, and password that you have been provided with:

```
# ---------------------- #
# On your local machine #
# ---------------------- #

whoami
# florent

uname -a
# Darwin BADWLRZ-AB12345 20.6.0 Darwin Kernel Version 20.6.0: Mon Aug 30 06:12:21
    PDT 2021; root:xnu-7195.141.6~3/RELEASE_X86_64 x86_64


ssh <user>@<IP>
# Replace <user> and <IP> with yours.

# The authenticity of host <IP> can't be established.
# ECDSA key fingerprint is SHA256:BZgws5BArCfwE6rDuN5i/aLgZMAKuC4si2D+ZuLN5gw.
# Are you sure you want to continue connecting (yes/no)? yes
# Warning: Permanently added <IP> (ECDSA) to the list of known hosts.
# <user>@<IP> password:
```

```
20
21 # Type your password when prompted, it's normal if you don't see what you type!
22
23 # ------------------------------ #
24 # On the compute cloud workbench #
25 # ------------------------------ #
26
27 # Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-81-generic x86_64)
28
29 whoami
30 # <user>
31
32 uname -a
33 # Linux containers-workbench 4.19.0-14-cloud-amd64 #1 SMP Debian 4.19.171-2
       (2021-01-30) x86_64 GNU/Linux
34
35 # Success! We can proceed from here!
```

Code snippet 1: We SSH into the cloud workbench

Bonus Question

Can you make the SSH login part simpler? For example, can you tweak your system such that the command: `ssh containers-workbench.cc.lrz.de` directly SSH into the machine for you?

Hint: You may want to create an SSH key and edit `~/.ssh/config` on your host system.

When you have verified that you indeed are into the cloud, let's see some Docker commands (you can use `man docker` to get the manual at any time):

```
1 # Docker requires privileges. Let's escalate now!
2 sudo -i
3
4 # Let's make sure docker is available
5 docker -v
6 # Docker version 20.10.18, build b40c2f6
7
8 #Let's see if there is a container running on the system
9 docker ps
10 # CONTAINER ID      IMAGE       COMMAND     CREATED     STATUS      PORTS
       NAMES
11
12 # ... No, nothing is running
13
14 # Let's see if we have some images already available on the system
15 docker image ls
16 #REPOSITORY     TAG     IMAGE ID        CREATED     SIZE
17
18 # ...Nope
```

Code snippet 2: Our first Docker commands

Looks like everything is fine! Enough about that, let's run our first container. You'll quickly understand why the Docker CLI has he reputation of being very intuitive. By default, the CLI is connected to the Docker hub and you can pull images on your computer, very much like you

would install applications on your phone via an App Store.

```
# No image are available. Let's pull one to test the installation
docker pull hello-world
# Using default tag: latest
# latest: Pulling from library/hello-world
# b8dfde127a29: Pull complete
# Digest: sha256:308866a43596e83578c7dfa15e27a73011bdd402185a84c5cd7f32a88b501a24
# Status: Downloaded newer image for hello-world:latest

docker run hello-world
# Hello from Docker!
# This message shows that your installation appears to be working correctly.

# To generate this message, Docker took the following steps:
#  1. The Docker client contacted the Docker daemon.
#  2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
#     (amd64)
#  3. The Docker daemon created a new container from that image which runs the
#     executable that produces the output you are currently reading.
#  4. The Docker daemon streamed that output to the Docker client, which sent it
#     to your terminal.

# To try something more ambitious, you can run an Ubuntu container with:
#  $ docker run -it ubuntu bash

# Share images, automate workflows, and more with a free Docker ID:
#  https://hub.docker.com/

# For more examples and ideas, visit:
#  https://docs.docker.com/get-started/
```

Code snippet 3: We pull the hello-world image and run the container

Question

Do you understand the output of the hello-world container? Is everything working well?

Good, this was our first container. Not very impressive? Let's run something interactively then!

```
docker run alpine
# Unable to find image 'alpine:latest' locally
# latest: Pulling from library/alpine
# 213ec9aee27d: Pull complete
# Digest: sha256:bc41182d7ef5ffc53a40b044e725193bc10142a1243f395ee852a8d9730fc2ad
# Status: Downloaded newer image for alpine:latest

# It pulled the image for us but didn't do anything?

# Let's see if the container is running
docker ps
# CONTAINER ID      IMAGE       COMMAND     CREATED     STATUS      PORTS
    NAMES

# Nothing is running !?

# Let's check stopped containers
```

```
docker ps -a
# 68febd19ee0b       alpine         "/bin/sh"      About a minute ago   Exited (0)
    About a minute ago    upbeat_knuth

# Nope, nothing more
# It successfully executed the command "/bin/sh" then exited. It's normal!

# Let's run it interactively, with the hostname 'bonjour' for example
# First check the name of the host machine
hostname
# course-node

# Run the container interactively
docker run -it --hostname "alpine-container" alpine

# Let's check if we are indeed running within an Alpine container, isolated from
    the host!
hostname
# alpine-container

# Let's see what processes are running in the container
ps aux
# PID   USER     TIME  COMMAND
#     1 root      0:00 /bin/sh
#     7 root      0:00 ps

# Not much is going on, that's quite minimalist.
# No OS is running (systemd, init...)
# Only our processes!

exit # You can also use ctrl-D
```

Code snippet 4: We interactively run an Alpine container

Question

Try common UN*X commands: `ls`, `cd`, `pwd`... Are some of them missing? Is it a bug or feature of containers? Of Alpine?

Going further: Use the `apk` package manager to find and install additionnal packages. How about: `man`, `python`, `htop`, `neofetch` for example.

Question

Explore the filesystem around you. Can you see files from the host system?

Create a file, for example with : `echo "Servus! Bonjour! ¡Hola! Hello!" > hello.txt`. Exit the container and start it again. Is the file still here?

Would you dare executing **`rm -rf /* in the container`**. What happens? Exit the container and start it again? What happened?

Hint: Remember that containers are ephemeral and disposable. Feel free to mess and break as many things as you can.

2 Under the Hood of Containers

Alright, things should get more or less clear for you and you should start to grasp how containers work. It is now a good time to dive deeper and see what they are made of, how they work, and what features they provide in order to be really useful.

2.1 The Bolts and Nuts of Containers

2.1.1 namespaces, cgroups, and copy-on-write Storage

Remember when we compared containers to VMs (section 1.3)? We saw that they are very similar on a high level: You can get a shell in them (shell exec, SSH...), you get your own process space, network interfaces, package manager, services... However, even if containers and VMs look, feel, and smell the same, they are very different on a low level approach. Containers are different than a full distinct machine, bu are instead a bunch of processes running on a normal kernel. they can't boot a different OS, don't have their own modules, don't need a PID 1, and don't need `syslogd`, `cron` etc.... And yet, they work, they are more efficient than VMs in many regards! Although containers may look quite magic, they are built on top of great old Linux features including: `namespaces`, `cgroups`, and `copy-on-write` Let's dive a tiny bit into kernel specifics of Linux here! Here are the low-level bolds and nuts that make containers work (fast!).

- `namespaces` are used to provide an isolated view of the system to processes running within the container. `namespaces` make it impossible for processes to escape the container and see what's happening elsewhere, be it the host or other containers running next to them. `namespaces` allow containers to have their own PID space, network interfaces, volumes, hostnames etc. Most importantly, `namespaces` allow a user to be root within a container while securely remaining a un-privileged user on the host (more about that in section 2.3).
- `cgroups` are used to control the hierarchical resource management and constraints, enforcing resources quotas (*e.g.,* CPU, RAM, I/O, bandwidth usage...) at the process level. This is particularly useful in an HPC context. `cgroups` can be enforced as soft or hard limits allowing to finely tune the resources access of the container. `cgroups` also come handy when migrating containers across compute nodes by providing "freeze" capabilities.
- `copy-on-write` storage makes possible to package containers into file system snapshots we call images. They are stored in a layer topology making it easy to pull, build, and extend existing images.

Nothing magic her: `namespaces`, `cgroups`, and `copy-on-write` are actually baked into the Linux kernel! Even without containers, the kernel is using namespaces and cgroups, you can see the system is a big container. *Even without containers, you're running into a container!* Repeat after me: "Containers are not a virtualisation technology. Containers are just normal processes isolated on the host. They don't come with a performance hit".

2.1.2 Volumes, Networking, and More

Let us now talk about the actual bolts and nuts you will actually tweak to make containers work for you: what is actionable for you behind your keyboard. We have seen that containers can conceptually be seen as VMs, and for good reasons: you can abstract volume storage, port mapping, environment variables etc. This becomes quite handy if you want to use persistent data

within your containers[5], or expose ports for your container to communicate with other services over the network. We can even add quotas and tune the resources the container can use on the host. We highlight these features in the snippets 5, 6, and 7.

```
# ----------- #
# ON THE HOST #
# ----------- #

# Let's create a sample directory on the host
# It will be mounted inside the container
mkdir /tmp/data
# Let's create some files we want to be persistently available within the
    container
echo "Monday, Tuesday, Wednesday" > /tmp/data/week.txt
echo "Jeudi, Vendredi, Samedi"    > /tmp/data/semaine.txt
echo "Sonntag, Sonntag"           > /tmp/data/woche.txt

# -v lets us mount volumes with the syntax: <path_on_host>:<path_in_container>
# When we run the container
docker run \
  -it \
  --rm \
  --name weeks \
  -v /tmp/data:/data \ # Host /tmp/data is mounted in /data in the container
  alpine

# Let's enter the container, interactively, with the "sh" shell
docker exec -it weeks /bin/sh

# ---------------- #
# IN THE CONTAINER #
# ---------------- #

ls -lah /data/
# total 20K
# drwxr-xr-x    2 root     root        4.0K Mar 26 10:07 .
# drwxr-xr-x    1 root     root        4.0K Mar 26 10:08 ..
# -rw-r--r--    1 root     root          24 Mar 26 10:07 semaine.txt
# -rw-r--r--    1 root     root          27 Mar 26 10:07 week.txt
# -rw-r--r--    1 root     root          26 Mar 26 10:07 woche.txt

# Success! The files are persistently written on the host drive,
# while being available within the container

exit
```

Code snippet 5: We map volumes between the host and the container

Question

Open a new tab and SSH into the host. Modify files in `/tmp/data`. Do the modifications appear in the container `/data` directory? What happens if you delete these files from the container, are they also deleted on the host?

One step further: Can you mount `/` into the container? Is it safe to even try?

[5] Remember earlier when you created a file in the container, stopped it, and started it again?

Figure 3: Demonstration of volume and port mapping with a container running a webserver

Bonus Question

Map the container directory `/usr/bin` to `/tmp/bin` on your host. Install `neofetch` within the container and execute it. Come back to your host and execute the same executable from the mapped volume. Are you surprised by the output? Can you explain why some things are the same and others not?

```
# Let's expose a service, for example a website!

# Create your website
echo "<h1>Welcome...</h1><p>...to my awesome website running in a container ;-)</p
    >" > index.html

# -d lets us run the container in detached mode (i.e., in the background)
# -p lets us map ports with the syntax: <port_on_host>:<port_in_container>
# -v lets us mount volumes with the syntax: <path_on_host>:<path_in_container>
docker run -d \
  --name webserver \
  -p 8888:80 \ # Host port 8888 is mapped to container port 80
  -v $PWD/index.html:/usr/share/caddy/index.html \
  caddy # caddy is a web server. Kinda like nginx or apache, but shinier!

# Visit http://<IP address>:8888 with your browser to see your website!
```

Code snippet 6: We map ports between the host and the container

Question

Can you use docker logs to see what's happening within the webserver?

```
# Let's cap the resources for the container: let's say 4 CPU, and 4GB RAM

docker run \
  -it \
  --cpus 4 \
  --memory 4096MB \
  alpine

```

```
# We can verify !

nproc
# 4
top
# Mem: 1135696K used, 9067380K free, 409600K shrd, 21492K buff, 675220K cached
```

Code snippet 7: We declare the resources allocated to the container

Now that we have seen the most important options, let's put them all together in this example:

```
# Let's suppose we want to run python code for big data analytics that makes use
    of a mongo database!

# -p lets us map ports with the syntax: <port_on_host>:<port_in_container>
# -v lets us mount volumes with the syntax: <path_on_host>:<path_in_container>
# -e lets us pass environment variables to the container
# -w lets us define the working directory when the container is started

# Let's run the database
# Persistence will be assured thanks to the -v flag
docker run -d --network pipeline \
  --name mongo \
  -e MONGO_INITDB_ROOT_USERNAME=mongoadmin \
  -e MONGO_INITDB_ROOT_PASSWORD=chang3me! \
  -v /home/user/pipeline/db:/data/db
  mongo

# Let's run the python code
# Assuming your code and config in the current directory
docker run -it --network pipeline \
  --name pipeline \
  -v "$PWD":/usr/src/myapp \
  -w /usr/src/myapp \
  --link mongo \
  python:3 python script.py
```

Code snippet 8: We put it all together

Question

Use the command `docker inspect` followed by the name of the container. What do you see? Can you find the information about mounted volumes? Can you find information about networking, environment variables, performances etc.?

2.1.3 Configuration and Image Creation

Good! So far we have been using pre-baked images provided by the Docker hub. How about we now look into how to create our own custom image. One solution would be take a base image, spin a container out of it, get into it interactively, and make all the changes we need (add environment variables, install packages, compile software...) and finally save (more precisely "commit") our container back to an image. We have seen that everything that happens into a container is not persistent, that's why the commit step is paramount. We showcase the process with the snippet 9.

```
# ---------- #
# On the host #
```

```
# ---------- #

docker image ls
# REPOSITORY          TAG                 IMAGE ID            CREATED
    SIZE
# alpine              latest              28f6e2705743        5 weeks ago
    5.61MB

# Only alpine is available as an image
# Let's customize it in order to use it to extract exotic rar files
docker run -it alpine

# ---------------- #
# In the container #
# ---------------- #

# We add a package to the container
apk update && apk add unrar
# ctrl-P ctrl-Q

# ---------- #
# On the host #
# ---------- #

docker ps
# CONTAINER ID        IMAGE               COMMAND             CREATED
    STATUS              PORTS               NAMES
# 5a1c7e2f8491        alpine              "/bin/sh"           59 seconds ago
    Up 57 seconds                           musing_kilby

# We want to commit the container 5a1c7e2f8491
docker commit -m "unrar capability added to the container" 5a1c7e2f8491 unrar-
    apine
# sha256:e5e572c22a2c84ebcb07c963203c07f89a4b47f848aceb4d80be8afbb884fa3e

docker image ls
# REPOSITORY          TAG                 IMAGE ID            CREATED
    SIZE
# unrar-apine         latest              e5e572c22a2c        55 seconds ago
    9.67MB
# alpine              latest              28f6e2705743        5 weeks ago
    5.61MB

# A new image is created, we can now run alpine container withthat contains unrar!
```

Code snippet 9: We commit a container to an image. It is a tedious process

I hope that from this example you see that it is a manual, cumbersome, and error prone process. Therefore, it would be nice to have the ability to simply describe how the final image should look like and let Docker build it for us. Using a Domain-Specific Language (DSL) will make things way easier and reproducible. The DSL of choice for building images is the Dockerfile. It is important to understand and master it as it will become the cornerstone of your container workflow, wether the image you create is meant run with Docker, enroot, or Charliecloud. We show on the code snippet 10 a basic Dockerfile that will build the same image as before but this time, automagically!

```
# We create the file "Dockerfile" with the content:

FROM alpine:latest              # The base image to extend
```

```
4 RUN apk update && apk add unrar # Run command within the container
5 VOLUME data_to_unrar            # Describe the volume
6 ENTRYPOINT /usr/bin/unrar       # The command to execute when the container is
      started
7
8 # Then build image with:
9 # docker build . -t "alpine_unrar"
10 ## . means that the Dockerfile is in the current directory
11 ## -t tags the image with the name "alpine_unrar"
```

Code snippet 10: We discover the Dockerfile

You see that it is much easier to read and understand. It is also very portable: you can share this file and let others build the image themselves. It is also common practice to version control Dockerfiles with git for example. Whenever you want to create a image, think: Dockerfile! It allows you to extend a pre-existing image and build on its shoulder. Browse the Docker hub and you'll see the myriad of images ready to be extended, be it a Python environment, node runtime, database, or a plain Operating System like Alpine, Ubuntu, or Arch linux for example.

In the end, a Dockerfile is simply a sequence of instructions, read from top to bottom that will let you customize the base image it is inheriting via its first line and the `FROM` statement. Each line starts with a keyword (uppercase by convention) the most useful ones are:

- `FROM`: The base image to extend. They can be found on the Docker hub, other image registries, or you can even create your own. This keyword can only be found once in the Dockerfile, and on the first line.
- `RUN`: The command to run for the image creation. Imagine that you are into a container running from the base image and use it to execute commands to customise it (*e.g.,* `apt-get update`, `wget https://example.com/bin/binary`). Warning: commands are not run interactively, so remember to use appropriate flags to auto approve the commands (*e.g.,* `apt install -y python3-keras`)
- `COPY`: Copy file(s) between the host and the image. You will use it to put additional content inside the image (*e.g.,* code, resources, assets...).
- `ENV`: Set environment variables within the image.
- `VOLUME`: Define volumes to map between host and container. They will need to be mounted when starting the container with the `-v` flag.
- `EXPOSE`: Define the ports to expose from within the container. They will need to be mapped when starting the container with the `-p` flag.
- `WORKDIR`: Define the working directory for the commands to follow.
- `ARG`: Define variables to be used elsewhere in the Dockerfile, or that can be passed via the Command Line Interface when building the image. This can be useful to make the Dockerfile more flexible.
- `ENTRYPOINT`: Defines the command to be called when launching the container.

There are several additional keywords you can use. They can be found here. Don't worry, the best is to put things in practice to see how it works. Let's jump to the next hands-on. Very soon, you'll master the art of the Dockerfile.

2.2 Warm-up lap: Make an Artificial Neural Network Dream in a Container

In order to set what we have seen above in action – but also to make something fun – we will take advantage of the seminal work of Google on making Artificial Neural Network (ANN)s dream [9]. While this requires quite a sophisticated setup, we will leverage containers to make it super easy. You will build a Dockerfile form scratch that will let you customize an Ubuntu base image by installing Python and the required dependencies. You will need to copy the code and config file into the image. You will also map a volume to save the output of the dreams and expose a port to access the interface to execute the code. Sounds like a lot of fun!

A Bit of Background

Inspired by the human brain, Warren S. McCulloch (neuroscientist) and Walter Pitts (logician), see figure 4 developed the first concept of an ANN in 1943 [10]. The basic idea is to have a "neuron" that lives in a network, receives inputs, processes them, and generates an output. An ANN is a connectionist computational system made of neurones that processe information collectively, in parallel throughout a network organized in layers. An ANN is adaptive and has the ability to learn by changing its internal structure based on the information flowing through it. This is achieved by tuning weights, the number that controls the signal between two neurons [11]. Today, ANN are used to perform "easy-for-a-human, difficult-for-a-machine" tasks like optical character recognition, image classification, and speech and facial recognition for example.

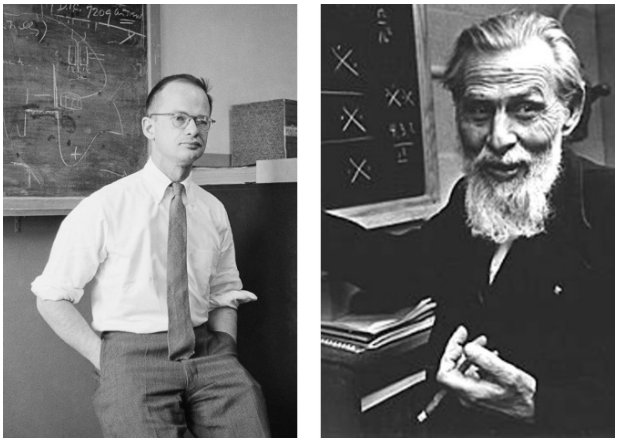

Figure 4: Walter Pitts (left) and Warren S. McCulloch (right)

The motivation behind Google's work on inceptionism was to try to understand why some networks perform well and other don't at certain tasks. In other words, using ANN is great as long as it works, and works well. They thus tried to peek inside a network to see what happens in each layer. In the first place, they fed images to to a network such that each layer progressively extracts higher and higher-level features of the image. The final layer finally decided what image to output. For example, the first layer maybe looks for edges or corners. Intermediate layers interpret the basic features to look for overall shapes or components (like a door or a leaf). The final few layers assemble those into complete interpretations. One way to visualize what goes on is to turn the network upside down and ask it to enhance an input image in such a way as to elicit a particular interpretation. Say you want to know what sort of image would result in "Banana": Start with an image full of random noise, then gradually tweak the image towards

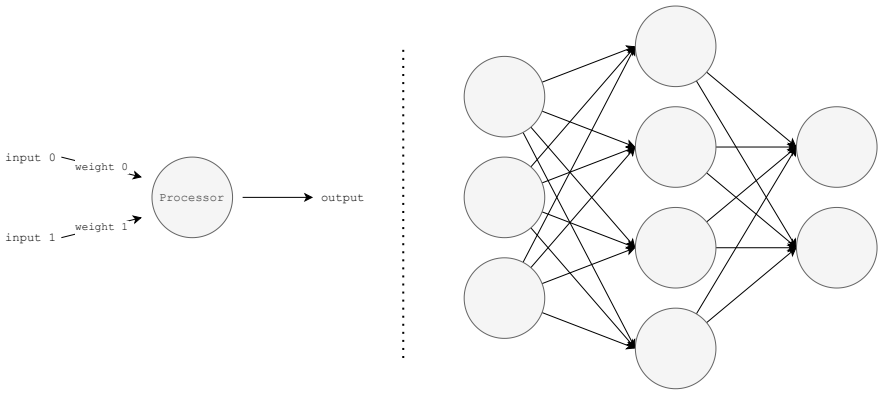

Figure 5: From the single neuron (left) to an Artificial Neural Network (right)

what the neural net considers a banana. By training networks by simply showing them many examples of what they want them to learn, hoping they extract the essence of the matter at hand (*e.g.,* a fork needs a handle and 2-4 tines), and learn to ignore what doesn't matter (a fork can be any shape, size, color or orientation), they successfully generated images out of noise like Bananas, Starfishes, Parachutes etc.

Later, instead of exactly prescribing which feature they wanted the network to amplify, they let it make that decision. They fed the network an arbitrary image or photo and let the network analyze the picture. They then picked a layer and asked the network to enhance whatever is detected. There you start to see what layer in the network is responsible for what feature. It appeared that lower layers tend to produce strokes or simple ornament-like patterns, because those layers are sensitive to basic features such as edges and their orientations. On the other hand, higher-level layers, which identify more sophisticated features in images, complex features or even whole objects tend to emerge. This approach creates a feedback loop and lead the network to recognize and enhance patterns in the image. Results are intriguing to say the least (see examples here) and have been quickly named "neural net dreams". Not only this work helped understand and visualize how neural networks are able to carry out difficult classification tasks, it also open the way for neural networks to become a tool for artists: a new way to remix visual concepts, or perhaps even shed a little light on the roots of the creative process in general [9].

Alright, that's cool, I'm sure you want to do it yourself with the image of your choice now!

Up to You Now

Question

The code to make the ANN dream is already written in a Jupyter notebook and provided to you on your machine. You want to run it in a container with a volume that exposes the base image for the ANN to dream, and port 8888 to access the web UI and run the code.
Create a Dockerfile that installs python and the required dependencies (TensorFlow, Numpy, Jupyter). Mount a volume in `/root/notebooks/data` and expose port 8888.

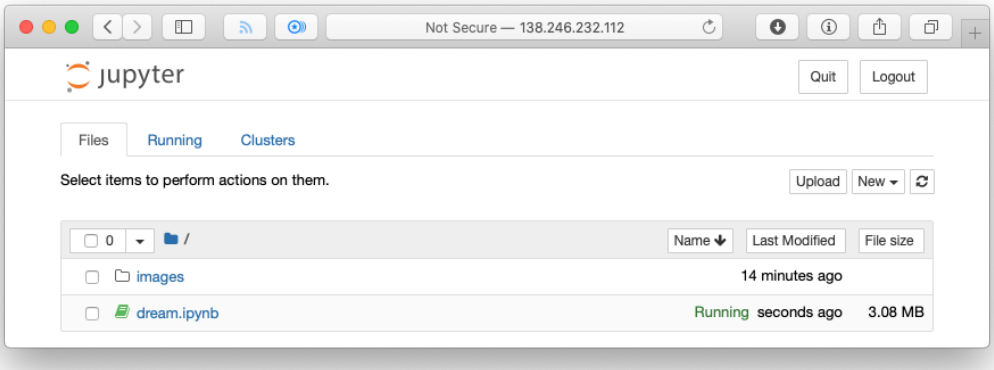

Figure 6: The Jupyter server is running

You'll know you got it right when navigating to `http://<your_ip>:8888` gives you the view depicted on figure 6. Help is provided on the next page if you need directions, try first without it!

A tiny “coup de pouce”

In order to get you started if you’re not sure how to proceed:

- Create an empty directory and `cd` into it.
- Create the file named `Dockerfile`.
- Create a file `requirements.txt` with the python dependencies (`tensorflow`, `matplotlib`, `jupyter`) listed on each line.
- Locate where is the folder that contains the python code as well as the jupyter config file.
- Edit the Dockerfile and extend an image easy to use, for example ubuntu, version bionic.
- Run the updates with apt and install `python3-dev` and `python3-pip`.
- Copy the requirements in the container
- Run the install of the requirements with `pip install -r requirements.txt`
- Copy the jupyter config in `/root/.jupyter/jupyter_notebook_config.py`
- Make /root/notebooks the working directory
- Copy the jypyter notebook
- Expose port `8888`
- Make the container start with the `entrypoint.sh` file provided to you
- Build the image and run the container with the volume `/root/notebooks/data` mounted

Going beyond

Too easy for you? How about you try:

- Changing the login password of the notebook
- Optimizing the image size following the best practices. How much lighter can you make it?
- Making the Dockerfile as small as you can. How short can you make it? What base image did you use?

2.3 Containers and Security

I hope that at this point you are excited about Docker and start to picture the realm of possibilities that are opening to you. But keep your head cool, there are still a couple of steps to go through before seeing your container with all your fancy AI pipelines running on top-tier HPC systems like the one provided at the LRZ. Indeed, while Docker is very powerful, it still has some

shortcomings in its implementation that make it impracticable when it comes to highly stringent environments. In the next two sections, we will highlight security and performances problems that make Docker unsuitable to the kind of workload you will want to run on HPC systems. But no worries, there are other container stacks more suited to HPC and now that you know how to use Docker, you already know how to use them.

Bonus Question

Use the docker bench security image to check the security of your current system. What do you see? Can you find warning related to the user namespace and to the docker socket?

Hint: You want to run:

```
docker run -it --net host --pid host --userns host --cap-add audit_control \
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /var/lib:/var/lib \
    -v /var/run/docker.sock:/var/run/docker.sock \
    -v /usr/lib/systemd:/usr/lib/systemd \
    -v /etc:/etc --label docker_bench_security \
    docker/docker-bench-security
```

Hint bis: During this course it's fine, but don't run random Docker commands on your system. Especially if it's from an untrusted source on the internet.

2.3.1 Security and User Namespace

Docker uses the namespaces to run containers (remember section 2.1.1). Let's see how it is actually on the system:

```
# Let's run a process in detached mode
docker run -d --name sleepy_container ubuntu:latest sleep infinity
# 60cf7524f7fe416b723722880b68da522a9fac7767c68de8874ec000ec5ced5f

# Let's see what's happening on the host
ps aux | grep sleep
# root       2846  0.0  0.0   2512   584 ?        Ss   08:36   0:00 sleep infinity
# root       2897  0.0  0.0   6144   880 pts/0    S+   08:37   0:00 grep sleep

# OH, the process is running as root (!!!!)
```

Code snippet 11: We explore the security of the user namespace with Docker

We see that the process is running on the host as root. The same goes for volumes, that's no good. That means that a rogue user may take advantage of the container and may bounce back on the host machine as root, mess with the volumes, and the processes. There is a workaround that consists in using user mapping in order to make the Docker daemon start containers with a specific, non-root user [12].

2.3.2 Security and Docker Socket Exploit

Docker's implementation used a daemon that runs on the host and manages images and containers. A listening socket allows it to remotely use the Docker processes running on the machine.

While this provides a lot of flexibility and makes the process of dealing with containers very easy for the end-user, it also come with some disadvantages. We show on snippet 11 the security hole engendered by the Docker socket.

```
# Let's run a simple ubuntu image with access to the socket
# We use -v to mount it
# You'll be surprised how often it is necessary

# ---------- #
# ON THE HOST #
# ---------- #

docker run -it --rm --name dind --hostname dind -v /var/run/docker.sock:/var/run/docker.sock ubuntu:18.04

# ---------------- #
# IN THE CONTAINER #
# ---------------- #

apt update
apt install docker.io -y

# The newly docker engine will use the socket piped into the host
docker ps
# CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
# f69fa8025f4e        ubuntu:18.04        "/bin/bash"         3 minutes ago       Up 3 minutes                            dind
# The container is seing itself from the outside

# How about we start a container within the container with elevated rights !?
docker run -d --name rogue --privileged ubuntu sleep infinity
docker exec -it rogue bash
# I am in a container from within a container that has privileged rights
# Now we can mess around

fdisk -l
# Disk /dev/vda: 40 GiB, 42949672960 bytes, 83886080 sectors
# Units: sectors of 1 * 512 = 512 bytes
# Sector size (logical/physical): 512 bytes / 512 bytes
# I/O size (minimum/optimal): 512 bytes / 512 bytes
# Disklabel type: dos
# Disk identifier: 0x00000000

# Device     Boot Start      End  Sectors Size Id Type
# /dev/vda1  *     2048 83886046 83883999  40G 83 Linux

# Ooops, I see what I shouldn't be able to see, disks of the host for example
# Let's mount the harddrive from the physical machine into the container
mkdir /mnt/host_HDD
mount /dev/vda1 /mnt/host_HDD
ls /mnt/host_HDD
# bin   dev  home        initrd.img.old  lib32  libx32      media  opt   root  sbin  sys  usr  vmlinuz
# boot  etc  initrd.img  lib             lib64  lost+found  mnt    proc  run   srv   tmp  var  vmlinuz.old

# Oooops, that works. I have in fact root access on the host
echo "You've been hacked" > /mnt/host_HDD/coucou.txt

```

```
# Your imagination is the limit now
# You can see the users, their files, etc.
# You can add your SSH key on the host and gain direct access
# You can download and run arbitrary software remotely
# You can encrypt the drive and ask for bitcoins
# ...
```

Code snippet 12: We exploit the Docker socket

The process was: On the host, launch a container and install docker in docker. Independently of the user who created the docker in docker container, you can create a privileged one. Since the socket is mounted as a regular volume, you had access to the host and were able to gain root access on the host. In this case it is a VM running the compute cloud of the LRZ but it may as well be an HPC system(!). You now understand why you won't see Docker on these systems!

2.4 Containers and HPC

Traditional HPC systems have narrowly focus software stacks that are operated in stringent conditions in terms of security and performances. Traditionally you had the following constraints when operating in an HPC environment:

- **Limited access to software libraries**: Are only made available on the system a subset of common libraries that have been reviewed and approved by the system administrators.
- **Limited rights**: You won't get root access on the system and will be unable to run arbitrary commands or customize your environment.
- **Limited connectivity**: Some HPC systems aren't even connected to the internet so don't expect to pull third-party components or assets.

HPC systems are inherently limited in doing a few things, but doing them very effectively in a massively optimized and distributed manner. It means that HPC systems were only used to run code implementing Message Passing Interface (MPI) along with source code you needed to compile yourself. Recent developments in Big Data analytics and AI/ML have shown how important it is for user to have access to cutting edge and exotic libraries, without needing the system administrators to review and approve all of them one by one. This is precisely where User Defined Software Stack (UDSS) comes into play. Containers can provide a securely isolated environment that run close to the host kernel, enabling high security and high performance. However, we also have seen that Docker is mostl business / consumer grade, and not so much HPC grade. Docker has some shortcomings regarding:

- **Performance:** Overlayfs has not been designed for performances and can be so slow that it becomes limiting for HPC.
- **Integrity:** File removal is done by "whiting out" which may cause subtile issues when dealing with large file or complex data structures or may require specific handling.
- **Associativity:** Between the process spawned by the docker daemon with the CLI that launched it needs to be solved making it hard to manage the workloads and allowing user to gain root access on the host. (not to mention the socket exploit showcased in snippet 12)

Many container environments have been developed since the introduction of Docker, some of them academia developed specifically for HPC environments. We show on table 3 that containers environments like enroot or Charliecloud fulfill our need for security and performances in HPC.

Table 3: Comparison of common User Defined Software Stack (UDSS)

Name	**Description**	**Strengths**	**Weaknesses**
Docker [2]	Historical and industry standard that provides the ability to package and run an application in an isolated environment.	Easy to use and widespread.	Not designed for security; conflicting `cgroups` with HPC environments; not optimized for performances
Singularity [13]	Developed at LBL to be a containerization solution for HPC systems. It enables Docker images to be converted into secure container images that can be run in userspace.	Supports several HPC components such as resource managers, job schedulers, and contains built in MPI features. Singularity can be used to run GPU-accelerated containers (see section 3.2)	Potential security issues came to light during a security review at LRZ in 2018, which have since been resolved in later versions. As a result of the internal security review and concerns of the system administrators. LRZ current policy does not allow to use Singularity.
enroot [14]	Developed by NVIDIA, allows to convert traditional Docker containers into unprivileged sandboxes. It Allows to run rootless containers without performance hit, making it particularly suited to HPC environments. It focuses on performance, portability, and reproducibity. It is the solution of choice at the LRZ.	Straightforward to use, provides built-in GPU support, no performance overhead	GPU support is limited to NVIDIA's architecture.

Continued on next page →

Table 3 – continued from previous page

Name	Description	Advantage	Drawback
Charliecloud [15, 16]	Developed at LANL to be a lightweight open source UDSS implementation based on the Linux user namespace for HPC sites with strict security requirements. It employs Docker to build the Charliecloud image, shell scripts to unpack the image to an appropriate location and a C program to activate the image and run user code within the image.	Charliecloud's distinct advantage is the separation of the build phase from the runtime and the usage of the newly introduced user namespace to enable non-privileged launch of containerized applications. The user namespace is an unprivileged namespace and within the user namespace, all other privileged namespaces are created without the requirement of root privileges, which means that a containerized application can be launched without requiring privileged access to the host system.	Less wideread than Singularity or enroot.
Shifter [17]	Shifter works by converting Docker images to a common format that can then be distributed and launched on HPC systems. Shifter works by enabling users to convert the Docker images to a flattened format, which are directly mounted on the compute nodes using a loopback device.	Good choice for conventional HPC batch queuing infrastructure	However, Shifter was developed to work well on the systems at NERSC and it does not appear to work as well on HPC systems at other centers out of the box. In addition, Shifter requires more administrative setup than other HPC container technologies.

Continued on next page →

Table 3 – continued from previous page

Name	Description	Advantage	Drawback
Podman [18]	Is an open source container management tool for developing, managing and deploying containers on Linux systems.	Runs without any additional permissions, just as a normal user process.	Requires user namespaces : it is as secure as any other process running in a user-namespace in linux; not as widespread as other UDSS and not oriented towards HPC.

Let us give you an example of how a container workflow looks like with enroot. It is super easy.

```
# Let's not be root, like on an HPC system
su ubuntu

# Let's work in a directory
mkdir "enroot" && cd $_

# We import the tensorflow Docker image as a squash filesystem
enroot import docker://tensorflow/tensorflow

# We create an enroot image out of it
enroot create tensorflow+tensorflow.sqsh

# And start it
enroot start tensorflow+tensorflow

# Seem like not much happened but we are in a container !
# Let's verify we indeed have tensorflow available
python3
# Python 3.6.9 (default, Jan 26 2021, 15:33:00)
# [GCC 8.4.0] on linux
# Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> print(tf.__version__)
# 2.6.0

# It works
# Now, let's see if we can be root in the container !

exit
# exit

enroot start --root tensorflow+tensorflow

whoami
#root

# It works ! I can have access to everything in the container!
# But am I root on the host ?? Hopefully not !!
sleep infinity

# Going back to the host
ps aux | grep sleep
# ubuntu     13194  0.0  0.0   6284   856 pts/0    S+   18:44   0:00 sleep
    infinity
```

```
# ubuntu     13296  0.0  0.0   8160   740 pts/2    S+   18:44   0:00 grep --color=
    auto sleep

# In fact I am still ubuntu on the host ! Success !
```

Code snippet 13: We are root without being root with enroot

2.5 Flat-out: Containers and HPC AI With enroot

In this hands-on, we want to show you how you can leverage your new experience with Docker to run high performance, hardware accelerated enroot containers. When operating on the LRZ AI systems, these containers will have direct access to Graphics Processing Unit (GPU) resources. You'll see that with the knowledge and the experience you gained with Docker makes up for 99.99% of the effort.

We'll use speech recognition inference in this exercise, this is a hard problem that AI and containers make surprisingly easy and fast to use. Historically, there was a presumption that resolving the inherent ambiguity in spoken language necessitated a deep understanding of the speaker's intent. Early speech recognition systems, exemplified by the 1990s program Dragon-Dictate, were rooted in specialized linguistic knowledge. They incorporated insights from syntax, grammar, phonetics, and even physiological constraints related to human vocalization. These systems relied on techniques such as hidden Markov models, spectral analysis, and cepstral compensation, often requiring users to pause between spoken words. Not only they were quite complex program, they were also quite expensive, count $9'000 for the first version of Dragon-Dictate [19]

With the advent of neural networks, the approach to speech recognition began to shift. The prevailing sentiment in the research community gravitated towards creating A.I. systems that could emulate human learning processes rather than merely encapsulating static information. This perspective marked a critical transition in the field. As Rich Sutton wrote in his Essay, "The Bitter Lesson", the goal of A.I. research should be to build "agents that can discover like we can" rather than programs "which contain what we have discovered" [20]. This is the bitter lesson, don't try to be smart about what you do, throw compute and data at a model instead.

In a strategic move, OpenAI open-sourced the Whisper project and model, providing both the code and a detailed explanation of its architecture. Whisper allows you to turn any speech into text and much more, like translating live etc [21]. Open sourcing Whisper, let to the development of many independant projects, most notably "Whisper.cpp" tht Georgi Gerganov, a programmer without extensive background in A.I. developed within a five-day timeframe, this standalone code consisted of 10,000 lines without requiring external dependencies.

For this hands-on, we'll run Whisper.cpp in a enroot container and convert some speech to text very easily. We need a Dockerfile that will build Whisper.cpp for us:

```
FROM alpine:3 AS builder
ARG WHSIPER_VERSION="v1.4.0"
RUN apk update; apk add --no-cache git make gcc g++
RUN git clone --depth=1 --branch=${WHSIPER_VERSION} https://github.com/ggerganov/
    whisper.cpp.git /opt/whisper.cpp
WORKDIR /opt/whisper.cpp
RUN make

```

```
FROM alpine:3
RUN apk update; apk add --no-cache gcc
COPY --from=builder /opt/whisper.cpp /opt/whisper.cpp
WORKDIR /data
ENTRYPOINT ["/bin/sh"]
```

Code snippet 14: The Dockerfile that allows us to run whisper.cpp

We need to build the image, import with enroot, create an enroot container and start it. As an example, we have some audio files and a model under `/data`, we'll need to mount this volume in the container. Therefore, the steps are the following:

```
# We build the Docker image
docker build . -t "whisper.cpp:1.4.0"

# We import the Docker image  into enroot
enroot import --output whisper.cpp:1.4.0.sqsh dockerd://whisper.cpp:1.4.0

# We create a container
enroot create --name whisper whisper.cpp:1.4.0.sqsh

# We can start the enroot container
enroot start --root --rw --mount /data:/data whisper

# There is a model and audio files in my /data folder
ls -lah
total 170M
drwxr-xr-x    2 nobody   nogroup     4.0K Oct 22 15:22 .
drwxrwxr-x   20 root     root        4.0K Oct 23 16:04 ..
-rw-r--r--    1 nobody   nogroup    23.7M Oct 22 15:22
    Albert_Camus_Discours_Prix_Nobel_1959.wav
-rw-r--r--    1 nobody   nogroup     4.9M Oct 22 15:22 Aphex_Twin_1993.wav
-rw-r--r--    1 nobody   nogroup   594.2K Oct 22 15:22 Me_at_the_zoo_2005.wav
-rw-r--r--    1 nobody   nogroup   141.1M Mar 22  2023 model.bin

# We can transcribe text!
whisper -m ./model.bin Me_at_the_zoo_2005.wav
# whisper_init_from_file_no_state: loading model from './model.bin'
# whisper_model_load: loading model
# whisper_model_load: n_vocab       = 51865
# whisper_model_load: n_audio_ctx   = 1500
# whisper_model_load: n_audio_state = 512
# whisper_model_load: n_audio_head  = 8
# whisper_model_load: n_audio_layer = 6
# whisper_model_load: n_text_ctx    = 448
# whisper_model_load: n_text_state  = 512
# whisper_model_load: n_text_head   = 8
# whisper_model_load: n_text_layer  = 6
# whisper_model_load: n_mels        = 80
# whisper_model_load: ftype         = 1
# whisper_model_load: type          = 2
# whisper_model_load: mem required  =  310.00 MB (+    6.00 MB per decoder)
# whisper_model_load: adding 1608 extra tokens
# whisper_model_load: model ctx     =  140.60 MB
# whisper_model_load: model size    =  140.54 MB
# whisper_init_state: kv self size  =    5.25 MB
# whisper_init_state: kv cross size =   17.58 MB
#
# system_info: n_threads = 1 / 1 | AVX = 1 | AVX2 = 1 | AVX512 = 0 | FMA = 1 |
    NEON = 0 | ARM_FMA = 0 | F16C = 1 | FP16_VA = 0 | WASM_SIMD = 0 | BLAS = 0 |
    SSE3 = 1 | VSX = 0 | COREML = 0 |
#
```

```
48 # main: processing 'Me_at_the_zoo_2005.wav' (304216 samples, 19.0 sec), 1 threads,
       1 processors, lang = en, task = transcribe, timestamps = 1 ...
49 #
50 #
51 # [00:00:00.000 --> 00:00:04.000]   Alright so here we are one of the elephants.
52 # [00:00:04.000 --> 00:00:12.000]   The cool thing about these guys is that they
     have really, really, really long punks.
53 # [00:00:12.000 --> 00:00:14.000]   And that's cool.
54 # [00:00:14.000 --> 00:00:18.000]   And that's pretty much all there is to say.
55 # [00:00:18.000 --> 00:00:18.840]   just say.
56 #
57 #
58 # whisper_print_timings:     load time =   237.47 ms
59 # whisper_print_timings:     fallbacks =   0 p /   0 h
60 # whisper_print_timings:      mel time =  1675.82 ms
61 # whisper_print_timings:   sample time =   101.14 ms /    80 runs (    1.26 ms per
       run)
62 # whisper_print_timings:   encode time = 10859.40 ms /     2 runs ( 5429.70 ms per
       run)
63 # whisper_print_timings:   decode time =  1059.07 ms /    78 runs (   13.58 ms per
       run)
64 # whisper_print_timings:    total time = 14074.81 ms
```

Code snippet 15: We run whisper.cpp into an enroot container

Question

Can you convert each wav file into text. Which one takes longer, why?

Going further: Can you convert the speech Albert Camus and translate it from French to English on the fly?

3 Containers on Nitromethane

In this section we present some of the powerful capabilities of containers. This is a bit of an extra since only an understanding of Docker, enroot, and Charliecloud is required for this module. At the end of this section, we will launch a complete transcriptomic pipeline with the help of a containers and you'll realize that they become very powerful when they actually disappear. We could have even started this course with the last hands-on so much it is easy!

3.1 Abstract Your Containers

When we say abstracting containers, we mean making them disappear. Containers true power arises when they serve a specific purpose without having to do any heavy lifting. In this section, we cover two fields of interest for us that benefit abstraction with containers, namely: Continuous Integration / Continuous Deployement (CI/CD) and reproducible scientific pipelines.

3.1.1 Continuous Integration / Continuous Deployment

We managed to cover quite a lot in the previous sections. However, I'm afraid the problem we initially presented on shipping software still stands. We haven't seen any actionable strategies you may use as a developer to safely push your code to production.

Idea #1: Ship an image: We've seen that we can package our application in a very effective manner using containers. We also have seen that containers can be committed and published as images to repositories like the Docker Hub for example. If you want to ship your code in a container, that would mean creating a Docker file, building the image, pushing this image to a repository (public or private), and let the Ops run this image as a container on their servers. That's a good improvement from the initial situation, but let's push a bit further. Some fair amount of overhead remains.

Idea #2: Use containers to continuously integrate and deploy your code: As you can expect from the title of the section, this idea is perhaps the best. The principle is never touching a container, but rather let them handle your code: building, testing, packaging (Continuous Integration) before automatically deploying it on the server (Continuous Deployment). It is specifically called continuous as it as happens on the fly, when the developer writes code. Leveraging CI/CD makes the developer a new kind of person. A developer v2.0 that also do Ops, or a "DevOps" for short. CI/CD bridges the gap between developers and Ops by letting the former define how the code is integrated and deployed thanks to containers. Containers are usually "workers" plugged into a Version Control System like git for example. Gitlab CI and Jenkins are two examples of tools enabling CI/CD with containers.

Now picture yourself as a newly stamped DevOps. You locally develop on your machine and your code is under git version control. When you commit your code, it is automatically remotely compiled and tested. Tag a commit, or push to a specific branch and here it is automatically in test or production, with the changelog parsed and release notes automatically generated on the release page. Nothing to worry about anymore. You're not even handling images or containers, just your code. Containers disappear. You could be coding on the beach, sipping a Piña Colada, you just have to push a few bytes on a cellular connection, and containers will have your back and take care of the rest.

3.1.2 Reproducible Scientific Pipelines

That sounds pretty cool but how does this translate to science? If you think bout it, if containers allow reproducible builds, they may as well be used for reproducible scientific pipelines. We know that data has never been more important in research and open data is becoming more and more of thing. Along with their research paper, scientist have been publishing their dataset and code for several years now. However, reproducing the results other researchers have published (or even your own(!)) can be surprisingly challenging [22, 23]. Scientific pipelines usually consists of many scripts that each call a specific library and generate a comprehensive report. It sounds reasonable to make use of containers to execute this code in a bundled and versioned environment. Simply pipe your data into it and comes the report out. Containers are agnostic of the underlying system and are the perfect candidate for reproducibility. We see here that containers become a piece in a data analysis pipeline. Once again abstracting them allow us to benefit from their qualities without having to deal with their ins-and-outs.

Nextflow is a good example of tool that allows scientists to create "data driven computational pipelines". With Nextflow, containers (Docker, Singularity, or even Charliecloud) simply become *providers*. Images containing a tagged version of a pipeline is pulled (see some of them here on the docker hub) and applied to the data. Nextflow comes with a DSL that lets scientist describe what they want to perform, and again, containers take care of the rest. We will experiment with Nextflow in the next hands-on, page 32. You will see that containers are completely abstracted and you would not even know that a docker container is running in the background!

3.2 Hardware Accelerated Containers

Although Central Processing Unit (CPU) performances kept increasing (and more or less according to Moore's law [24] from the 70's through the 90's), Graphics Processing Unit (GPU) acceleration remains the best way to accelerate resource intensive tasks. This is particularly in science and research where ever increasing workloads need to be processed at large scale (*e.g.,* simulations, model training, accuracy constrained workloads, data analytics etc.). PCIe passthrough is a solution that is commonly used to make hardware resources available to machines virtualized on a host. However, this trick doesn't scale very much with the number of VMs on a host nor with the number of hosts in a data center. Passthrough also often requires exotic configuration and rights escalation, raising maintenance and security burdens [6]. We've seen that one of the advantages of containers is that they run closely to the kernel and the metal. Therefore, passthrough is irrelevant and scaling GPU accelerated workloads with containers can be made easy, especially compared to a bare metal installation. Container gap several challenges commonly encountered in HPC [25]:

1. **Insufficient installation instructions:** It is such an niche field of application that such applications are usually built by a researcher or at best a research group. It usually focuses on addressing their specific needs and then open sourced the project to benefit like-minded researchers. However, these codes may have insufficient documentation complicating the job of the system admins (who don't have keen knowledge of these applications) to identify the software stack requirements, the installation process, and to run the debugging and testing. Too often, system admins are frustrated with an installation failure error arising from a missing dependency or version mismatch. This of course impact you by limiting your access to such resources.

2. **Multiple application dependencies:** Installing an application with a laundry list of libraries, drivers, and compilers adds complexity[6]. Furthermore, installing this on a shared system that has hundreds of other applications multiplies the complexity because different applications may require a different version of the same library and upgrading to the latest version of the library may create issues that prevent an installed application from properly executing.

3. **Rare application upgrades:** System admins, and often application users, delay upgrading an installed application to the latest version because it may make the system unstable and/or prevent the application from running altogether. As a result, users miss out on new features and optimized performance.

4. **Lack of performance reproducibility:** HPC researchers publishing their scientific papers consistently experience lack of reproducibility of their simulation results because performance on two identical systems will vary due to the software dependencies installed on them.

5. **Lower productivity:** Typically, users have to request system admins to install applications which may take a few days. This is counterproductive for both the system admin as well as the end user. Empowering users to deploy their applications on a shared resource enables resource-constrained system admins to focus on addressing mission critical tasks and improving user satisfaction.

From this enumeration of problematics, and according to the content of this document, you will agree that it looks like containers can address a lot (if not all) of the aforementioned challenges [25]:

1. **Simple application deployment:** Containers eliminate the need to install applications and allow users to pull and run containers on the system without asking the system administrator to intervene. Deploying an application takes just a few minutes, saving time for both users and administrators.

2. **All the dependencies in one place:** Containers include all the software required to run the application, delivering a great user experience. Furthemore, system admins do not have to worry about compatible software and versioning requirements.

3. **Access to the latest features:** Since containers are fast to deploy and won't create conflicts, users can easily access the latest versions of applications, the latest features and optimized performance.

4. **Portability and performance reproducibility:** Containers can be deployed on any system with a container runtime allowing users to test their simulations on various platforms. Since the software required to run the application is unchanged, users can expect the same performance across similar system configurations.

5. **Higher productivity:** Containers empower users to deploy applications on a shared resource without putting in a ticket to install applications. This cuts down deployment process from days to minutes, enabling users to run simulations sooner and system admins to focus on providing users a better data center experience.

[6] At this stage of the course, I hope when you see a sentence like this, the little "container light" in your brain is blinking.

6. **Performance similar to bare-metal:** Running applications from containers has little effect on performance.

An example of GPU accelerated containers is the NVIDIA GPU Cloud (NGC) which is built on top of the Singularity UDSS. It is marketed as "The Simplest Way to Deploy HPC Applications on GPU-Accelerated Systems" [25]. Not only it provides the stack to run containers on GPU, but is also offer a registry from which to pull optimized and up-to-date libraries and resources.

Very much like Charliecloud, dealing with NGC images relies on converting Docker images to the appropriate format. They can later be executed on the host, use mounted volumes etc. Under the hood, these containers make use of driver which consists of both kernel-space and user-space elements that are matched, making kernel components available to the root-less user to make use of them. This type of container usually lives in a Simple Linux Utility for Resource Management (SLURM) environment and uses MPI for parallelization.

3.3 Orchestration and Scaling Across a Compute Cluster

Kubernetes is a cloud-native open-source system for deployment, scaling, and management of containerised applications. It provides clustering and file system abstractions that allows the execution of containerized workloads across different cloud platforms and on-premises installations. Kubernetes main abstraction is the pod. A pod defines the (desired) state of one or more containers *i.e.,* required computing resources, storage, network configuration. Kubernetes abstracts also the storage provisioning through the definition of one more more persistent volumes that allow containers to access to the underlying storage systems in a transparent and portable manner. Kubernetes is manly use for orchestration (yes, you can give a quick look back to section 1.2), it means o it becomes handy when running containers on a large scale. although you will be fine using Docker on your workstation and Charliecloud on the HPC system to run your workloads, a tool like Kubernetes will ease the life-cycle management, networking, and scalability of your system. This is somewhat what schedulers like SLURM are already doing in the HPC world, but here tailored for containers. While HPC workload managers are focused on running distributed memory jobs and support high-throughput scenarios, Kubernetes is primarily built for orchestrating containerized in the form of microservices, allowing load-balancing, self-healing and recovery after failure, monitoring, logging etc.[26] In general, you are not expected to get a Kurbernetes infrastructure going, there is a lot of heavy lifting and expertise required. Therefore, you'd rather take advantage of an existing infrastructure, like the one LRZ is currently prototyping!

3.4 Last Lap: Reproducible Transcriptomic Workflow With Containers and Nextflow

This hands-on closes this module on containers, and yet, you won't even notice that containers are running! Containers will only be used as *providers* and will be applied to a dataset. They encapsulate a version controlled pipeline that can be run at any scale, from your computer, to an HPC system. Containers are a tool for reproducibility and logic and data are completely decoupled, see figure 7.

Let's suppose that you are a health researcher. You have access to two populations of cells. The first is composed of healthy cells while the other is made of ill cells. The latter is ill because

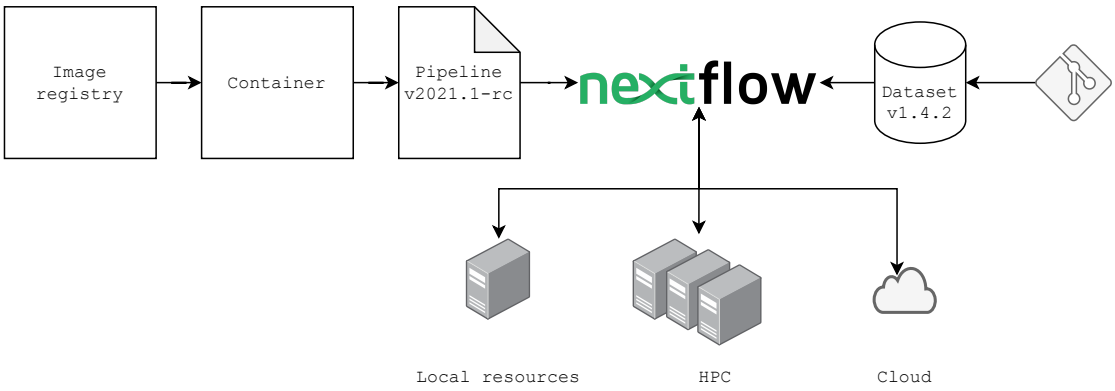

Figure 7: Nextflow leverages containers to allow separation of concerns

of a genetic mutation. In order to understand the disease, you must shed light on the genetic mechanism that is causing the difference between the two populations. You want to look at the difference in gene expression. When a gene (made of DNA) is active in a cell, RNA transcripts are made through the process named transcription. RNA-Seq makes use of high throughput sequencing to tell what genes are actives and how much they are transcribed. We measure the gene expression in both cell populations and compare the results to see what's happening in the mutated cells. A RNA-Seq experiment usually occurs in 3 main steps: i) Biological sample preparation (preparation of the library) ii) Sequencing iii) Data analysis. In order to prepare the RNA-Seq library, the RNA has been isolated, broken into small fragments (in order to allow sequencing[7]), and converted into cDNA (complementary DNA that is more stable and can be easily sequenced.). After adding sequencing adaptors to the cDNA, the fragments are amplified in for the sequencing to take place. Only then, high throughput sequencing can take place, for example using the commonly used Illumina technology. During sequencing, a score is associated with each base that is read telling how confident the sequencer is that the measure base is correct. Sequencing is not perfect. Therefore, the last step is primordial and will consists in quality control of the sequencing before performing any other analysis. The required steps are then: i) Filter out garbage reads ii) Align the quality reads to a genome iii) Count the number of reads per gene.

As you can expect from the nature of this hands-on, we will focus on the data analysis part! We will compare the transcriptome of two cell types located in the gut, and liver. Sequencing output is provided in FASTQ format (`.fq` files). We will run these analyses:

1. **FastQC:** To quality check the sequencing. Sequences with poor quality must be trimmed or filtered.
2. **MultiQC:** Also to quality check, with additional information.
3. **Salmon:** That must be run after the quality check and reads filtering. Salmon allows the mapping of high quality reads on a genome and a genes set in order to establish the differential gene expression.

[7]RNA transcripts can be thousands of bases long, while sequencers can only short fragments: 200-300bases.

```
# Let's get the Nextflow code along with the dataset
git clone "https://github.com/nextflow-io/rnaseq-nf.git"
#Cloning into 'rnaseq-nf'...

# And run the pipeline locally in a docker container
cd rnaseq-nf
nextflow run nextflow-io/rnaseq-nf -with-docker
# R N A S E Q - N F   P I P E L I N E
#  ===================================
#  transcriptome: /home/ubuntu/.nextflow/assets/nextflow-io/rnaseq-nf/data/ggal/ggal_1_48850000_49020000.Ggal71.500bpflank.fa
#  reads        : /home/ubuntu/.nextflow/assets/nextflow-io/rnaseq-nf/data/ggal/*_{1,2}.fq
#  outdir       : results
# executor >  local (6)
# [06/dd0ce9] process > RNASEQ:INDEX (ggal_1_48850000_49020000) [100%] 1 of 1 /
# [a5/514067] process > RNASEQ:FASTQC (FASTQC on ggal_liver)    [100%] 2 of 2 /
# [9c/af0a9d] process > RNASEQ:QUANT (ggal_liver)               [100%] 2 of 2 /
# [70/d1650e] process > MULTIQC                                 [100%] 1 of 1 /
# Done! Open the following report in your browser --> results/multiqc_report.html

# That's it ! The rport is published in the results folder!
# Use scp to get visualize it on your local machine
```

Code snippet 16: We run a reproducible transcriptomic pipeline with Nextflow

Biology Questions

- Look at the raw sequencing data. how much lines are required to give the output of a read? How long are these reads? Is it a lot?
- What sequencing technology as been used to generate the sequences used in this pipeline? Does it make sense with the size of the reads?
- Look at the fastQC report, where is the sequencing the better along reads? Where does the sequencer make error? Is it related to the sequencing technology you have identified above?
- Salmon is biased by the content in Kmers. Look at the reports, should we worry here?
- What statistical analysis would you run using the Salmon output provided by Nextflow.

Container Question

Can you use the caddy web server (as illustrated on code snippet 6) to expose the result of the pipeline as a web page accessible on port 8888?

One step further: What port should you use instead of 8888 if you don't want to specify it in the url? What additional port should you consider to allow encrypted connections via TLS? Is it possible on the current VM?

4 Take home message

Containers are:

- The ephemeral running instance of an environment: an application, its runtime, dependencies, libraries, settings etc.
- where processes are executed in isolation,
- thanks to a kernel feature called `namespaces`.
- They have limited access to resources thanks to another feature named `cgroups`.
- They run independently of the underlying infrastructure.
- They can be compared to Virtual Machine (VM)s but are fundamentally different.
- They indeed bundle a complete environment but no machine details.
- They run close to the metal and yield higher performances.
- They can mount volumes from the host and expose ports.
- Containers can be committed to images,
- manually or thanks to a Domain-Specific Language (DSL) like the Dockerfile.
- The syntax of the Dockerfile is expressive.
- The Dockerfile is the cornerstone of all container platforms.
- They are easy to write, share, and build.
- Once can convert a Docker image to other format like Charliecloud which is suited to HPC applications

Containers allow:

- The user to define the stack its application lives in: User Defined Software Stack (UDSS).
- An application to be easily portable and scalable.
- Consistency and reproducibility: in many regards, software build, scientific pipelines etc.
- You to spend more time on your code and your problematic, less on time consuming friction.

5 Recommendations

This section gives you alist of recommendations. Content of this course is loosely based on them. You may want to explore them to start over from the beginning or dive deeper in a specific aspect.

Books

- Container Security: Fundamental Technology Concepts That Protect Containerized Application
- Kubernetes 101

Blogs and tutos

- Simplilearn: Docker Tutorial For Beginners: A Step-by-Step Guide
- Play with docker

Podcasts

- Kubernetes Podcast
- DevOps and Docker Talk Podcast
- Discussing Docker Containers and Kubernetes with a Docker Captain

Channels and videos

- Docker run
- Docker from zero to hero

And the links in references, page 45

A Frequently asked questions

Why are containers called "containers" in the first place? When you face a problem, it is most likely that someone else before you had to deal with it. Shipping software is no exception. A a matter of fact, shipping physical goods is very similar and people have been facing similar issues. The shipping industry has been operating in the same way we use to ship software. You put your goods of all sorts (barrels, bags, furniture, boxes...) on an infrastructure (boat, train, horses...) necessary to ship it. Therefore, as a merchant, when you ship your goods, you have to worry about how it will be shipped (are my goods fragile? Will it be crushed by something else along the way? Does the staff will know how to handle my stuff?). This has been true for centuries. During the 50's, people gathered agreed on using a standard shipping unit to ease transportation and allow interoperaibility between the various infrastructure. They aggreed on the standard size, witght, how the doors open, what identification information to write on it... The intermodal container was born and changed the shipping industry by allowing separation of concern. From now on, if I want to ship something, I just have worry about what I put in the container and how. Once I close the doors, I don't have to worry about anything. The freight company will have their own set of conceerns which are independant of what I put in my container and how I arranged things inside. Seoaratiuon of concerns allowed automation and with automation comes reliability, reproducibility, decrease of costs... A software containers is a lot like a shipping container. As a developper, a container is a standadized way to take software component and deliver it to an infrastructure where the concerns of what's in the box: software stack, language... is irrelevant. In the same way a shipping container can be shipped accross the world independantly of the infrastructure to carry it, a software container allows software to run independently of the underlying infrastructure[2, 27].

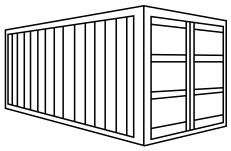

Figure 8: Representation of an intermodal shipping container

What is this layer thing Docker uses? Docker and other UDSS use copy-on-write strategies where images are built with successive layers. There are two types: i) Read-only (most of the layers) ii) Read-Write (top-most layer when we spin a container from the image). Images can share layers, making them lighter. Docker will be able to re-use a previously made layer from it cache to build a new layer. Considering a Dockerfile, a layer is created for each line starting with a Docker keyword. One can use `docker history <image:version>` to see the layers created to make the image. The `<missing>` layers are inherited from the parent image.In order to reduce the image size, consider wrapping multiple commands on a single line. When a container a running, one can use `docker diff <container>` to see modifications made on the writable layer. You should therefore be very careful not to add secrets during build phase as they will be able to be recovered in buried layers.

What are the implementation details of Docker and Charliecloud? Docker leverages the Kernel Namespaces feature offered by the Linux kernel in order to isolate processes in "containers". When creating a container, Docker associates all the namespaces to it in order to isolate the corresponding resources.

- **Namespace MNT:** For the mounting points of the file system: The container only sees its own mounting points and not the ones from the host

- **Namespace PID:** For the processes: The container can only access its own processes. It can not see what is happening on the host, nor in other containers. The processes numbering (PID) is initialized on the main processing within the container as PID 1.
- **Namespace NET:** For the network interfaces: The container can only see its own network interfaces. It can not see what is happening on the host, nor in other containers.
- **Namespace IPC:** For the communication between processes of type SysV: The container can make used of dedicated IPC channels and can not access the others on the host or in other containers.
- **Namespace UTS:** For the hostname and the NIS domain name: The container has its own NIS hostname can be different than the host or other containers.

Charliecloud uses a subset of these namespace in order to ensure security on the system and leverages the newly introduced user namespace to start processes without privileges.

B Best practices

B.1 When writing a Dockerfile

- Don't use `:latest` in image version: It will change and so may the behavior or your container.
- Leverage official images (cryptographically signed images etc).
- Chain commands in 1 RUN line
- Prefer `COPY` rather than `ADD` (more secure and reproducible).
- Use `--chown` in `COPY` commands
- Clean-up as you go: Clean temporary files, caches...
- Leverage multi stage build: separate compile and runtime (FROM ... AS build ... FROM ... CMD)
- Use new build kit
  - share ssh between host and container
  - `--mount=type=cache` to speed up build
  - Secret mount type
- Consider using tools to improve your images:
  - Hadolint: To lint your Dockerfile and implement best practices directly when writing the code.
  - Dive: To explore your images, reduce size and to make sure secrets are not kept in buried layers.

B.2 When using Charliecloud

- In your Dockerfile avoid building your application and copying/storing your data in /home, as when executing the Charliecloud container the users /home directory will map to the containers /home directory.
- It is easier to pull docker images and convert them
- Avoid copying too much stuff in the container, Charliecloud containers import the host systems environment (PATH, LD_LIBRARY_PATH) the user can use libraries, files and executable from the system. Just remember to mount these directories in the container (it is possible to unset variable though, using `--unset-env`).

B.3 Security Checklist for docker and other UDSS

We have discussed the security of container earlier (section 2.3). It is important to note that overall, containers are fairly secure to run application. The security of the system then depends much more on the host configuration and the images that are used[28].

On the host

1. Isolate the network of your host with the one from your containers
2. Keep the operating system and the container runtime up-to-date, specifically security updates.
3. Consider hosting sensitive containers on a dedicated host
4. Consider improving the security of the host by using SE Linux for example.

For the images

1. Do not blindly use an image downloaded from the internet. Read the Dockerfile and build it yourself if possible. Look at the scripts and annexed files.
2. Prefer to use official base image that are cryptographically signed by the developers.
3. Keep your images and packages up-to-date, specifically security updates.
4. Limit the amount of packages used in you image. Consider removing packets only needed for image creation (that can include compilers themselves).
5. When using a third party registry, very Docker content trust signatures
6. Do not store secrets in files or environment variables. Take advantage of the "Docker Secret" feature.

For the containers

1. Do not use the `--privileged` option or manually attach `/`, `/dev`, `/proc`, `sys` to the container.
2. If you really need to attach a device to the container, use `--device` and limit write capabilities
3. Isolate your containers on the network level if not necessary `--bridge=none`. `--network host` gives access to the network interface of the host to the container and should never be used
4. Remap user when starting a container to make sure root inside the container is not mapped with root on the host[12]
5. Containers should never be started with any of root capabilities. Starting containers with `--cap- drop=ALL` is safer.
6. It is safer to take advantage of `cgroups` to limit the the resources containers can access, and should not inherit control groups from the parent. Using `--memory`, `--memory-swap`, `--cpus`, `--cpu-period`, `--cpu-quota`to what is really needed is safer. Keep an eye on the resources usage.
7. When possible, it is safer to run a container with its root file system as read only
8. Monitor the container activity and export logs automatically for review

C Cheat sheets

C.1 Dockerfile

A Dockerfile cheat sheet that follows the good practices.

```
FROM repo.tld/maintainer/image:version

# This is a comment

LABEL maintainer     = "John Doe <john.doe@mail.com>"
LABEL institute      = "Leibniz Rechenzentrum"
LABEL de.lrz.version = "2022.2-rc"

ARG jdk_version=11

RUN apt-get update \
    && apt-get install -y --no-recommend \
        git \
        automake \
        build-essential \
        openjdk-${jdk_version}-jdk \
    && apt clean \
    && rm -rf /var/lib/{apt,dpkg,cache,log}/ \
    && rm -rf /var/cache/* \


WORKDIR /tmp
RUN git clone --depth=1 https://git.example.com/repo.git  \
    && cd repo \
    && ./autogen.sh && ./configure \
    && make && make install

COPY ./entrypoint.sh /entrypoint.sh

ENTRYPOINT /entrypoint.sh
```

This Dockerfile is built with the command: `docker build . -t name:version`

C.2 .gitlab-ci and Java application

This is mostly for reference and is not really part of the course. At least it showcases the best practices when writing a config file for Continuous Integration / Continuous Deployement (CI/CD)

```
image: maven:3-jdk-11

variables:
  NB_OF_WAR_TO_KEEP: 3
  MAVEN_OPTS: "-Dhttps.protocols=TLSv1.2 -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/
    repository -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.
    Slf4jMavenTransferListener=WARN -Dorg.slf4j.simpleLogger.showDateTime=true -
    Djava.awt.headless=true"
  MAVEN_CLI_OPTS: "-s $CI_PROJECT_DIR/../$CI_PROJECT_NAME.tmp/MAVEN_SETTINGS_XML
    --batch-mode --errors --show-version $MAVEN_ADDITIONAL_CLI_OPTS"

cache:
  paths:
    - .m2/repository

```

```
.set_build_infos_variables: &set_build_infos_variables
  - build_group=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}groupId').text.encode('utf-8')")
  - build_artifact=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}artifactId').text.encode('utf-8')")
  - build_properties_revision=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}properties/{http://maven.apache.org/POM/4.0.0}revision').text.encode('utf-8')")
  - build_properties_changelist=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}properties/{http://maven.apache.org/POM/4.0.0}changelist').text.encode('utf-8')")
  - "[ -z $CI_COMMIT_TAG ] && build_version=$build_properties_revision.$CI_COMMIT_SHORT_SHA$build_properties_changelist || build_version=$build_properties_revision"
  - build_packaging=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}packaging').text.encode('utf-8')")
  - build_description=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}description').text.encode('utf-8')")
  - build_name=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}name').text.encode('utf-8')")
  - build_filename=$build_artifact-$build_version.$build_packaging
  - repository_url=$(python -c "import xml.etree.cElementTree as ET; print ET.parse('pom.xml').find('./{http://maven.apache.org/POM/4.0.0}distributionManagement/{http://maven.apache.org/POM/4.0.0}repository/{http://maven.apache.org/POM/4.0.0}url').text.encode('utf-8')")
  - "[ -z $CI_COMMIT_TAG ] && WAR_FILE=target/$build_filename || WAR_FILE=.m2/repository/${build_group//\\./\\/}/$build_artifact/$build_version/$build_filename"

.tests-job-template:
  artifacts:
    reports:
      junit:
        - target/surefire-reports/TEST-*.xml
        - target/failsafe-reports/TEST-*.xml

build:
  extends: .tests-job-template
  stage: build
  script: "mvn $MAVEN_CLI_OPTS -Dsha1=.$CI_COMMIT_SHORT_SHA verify"
  artifacts:
    expire_in: 1 day
    paths:
      - target/*.war
  except:
    - tags

release:
  extends: .tests-job-template
  stage: build
  script:
    - "mvn $MAVEN_CLI_OPTS -Dchangelist= deploy"
    - *set_build_infos_variables
    - "CHANGELOG=$(sed -n \"/^## \\[${build_version//\\./\\\\\\.}\\]/,/^## / {/^## / d;s/\\r*$/\\\\\\\\n/;p}\" CHANGELOG.md | tr -d '\\n')"
```

```
    - "DESCRIPTION=\"$build_description\\\\n\\\\n## Changelog\\\\n$CHANGELOG\""
    - ARTIFACT_URL=$repository_url/${build_group//\./\/}/$build_artifact/$build_version/$build_filename
    - "curl --request POST
      --header 'Content-Type: application/json'
      --header \"Private-Token: $API_TOKEN\"
      --data \"{
        \\\"name\\\": \\\"$build_name v$build_version\\\",
        \\\"tag_name\\\": \\\"$CI_COMMIT_TAG\\\",
        \\\"description\\\": \\\"${DESCRIPTION//\\\"/\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\}\\\\\",
        \\\"assets\\\": { \\\"links\\\": [{ \\\"name\\\": \\\"$build_filename\\\",
     \\\"url\\\": \\\"$ARTIFACT_URL\\\" }] }
      }\"
      $CI_API_V4_URL/projects/$CI_PROJECT_ID/releases"
  only:
    - tags

.tests-job-template:
  services:
    - name: selenium/standalone-chrome:latest
  variables:
    START_XVFB: "false"
    SELENIUM_REMOTE_URL: "http://selenium-standalone-chrome:4444/wd/hub"
  resource_group: selenium-standalone-chrome

.deploy-war-job-template:
  stage: deploy
  script:
    - *set_build_infos_variables
    - mkdir -p ~/.ssh
    - printf "%s" "$SSH_PRIVATE_KEY_BASE64" | base64 --decode > ~/.ssh/id_ed25519
    - chmod 600 ~/.ssh/id_ed25519
    - printf "%s" "$SSH_KNOWN_HOSTS" > ~/.ssh/known_hosts
    - shopt -s expand_aliases
    - touch ~/.ssh/config
    - alias ssh="ssh -F ~/.ssh/config"
    - scp $WAR_FILE $SERVER_CONN:~/work/webapps
    - ssh $SERVER_CONN "cd ~/work && (./stop.sh; rm -rf app-link.war tomcat/webapps/ROOT; ln -sn webapps/$build_filename app-link.war && ./start.sh)"
    - ssh $SERVER_CONN "ls -tp ~/work/webapps/*.war | grep -v '/$' | tail -n +$(($NB_OF_WAR_TO_KEEP+1)) | tr '\n' '\0' | xargs -0 -r rm --"
  when: manual
  only:
    variables:
      - $SERVER_CONN
      - $SSH_PRIVATE_KEY_BASE64
      - $SSH_KNOWN_HOSTS

.deploy-from-artifact-or-rebuild-job-template:
  extends: .deploy-war-job-template
  before_script:
    - *set_build_infos_variables
    - "[ ! -f $WAR_FILE ] && mvn $MAVEN_CLI_OPTS -Dsha1=.$CI_COMMIT_SHORT_SHA -Dmaven.test.skip=true package"
  except:
    refs:
      - tags

.deploy-from-repo-job-template:
  extends: .deploy-war-job-template
  before_script:
    - *set_build_infos_variables
```

```
107     - "[ ! -f $WAR_FILE ] && mvn $MAVEN_CLI_OPTS dependency:get -
        DremoteRepositories=$repository_url -Dartifact=$build_group:$build_artifact:
        $build_version:war -Dtransitive=false"
108   only:
109     refs:
110       - tags
111
112 deploy snapshot on test:
113   extends: .deploy-from-artifact-or-rebuild-job-template
114   environment: test
115   resource_group: deploy-test
116
117 deploy snapshot on production:
118   extends: .deploy-from-artifact-or-rebuild-job-template
119   environment: production
120   resource_group: deploy-production
121
122 deploy on test:
123   extends: .deploy-from-repo-job-template
124   environment: test
125   resource_group: deploy-test
126
127 deploy on production:
128   extends: .deploy-from-repo-job-template
129   environment: production
130   resource_group: deploy-production
```

Code snippet 17: Example of base code to enable Java CI/CD on gitlab with containers

References

[1] Programmer Humor. *r/ProgrammerHumor - It works on my machine...* reddit. Aug. 27, 2019. URL: https://www.reddit.com/r/ProgrammerHumor/comments/cw58z7/it_works_on_my_machine/ (visited on 03/22/2021).

[2] dotconferences. *dotScale 2013 - Solomon Hykes - Why we built Docker*. 2013. URL: https://www.youtube.com/watch?v=3N3n9FzebAA (visited on 02/02/2021).

[3] Jacob Howard. *Repurposing Container Technologies for Development*. Docker Community All Hands Meeting. Online, Mar. 11, 2021. URL: https://havoc.io/talks/repurposing-containers (visited on 03/15/2021).

[4] Ben De St Paer-Gotch. *Tech Preview: Docker Dev Environments - Docker*. Docker blog. June 23, 2021. URL: https://www.docker.com/blog/tech-preview-docker-dev-environments/ (visited on 09/21/2022).

[5] *What is a Virtual Machine?* VMware. URL: https://www.vmware.com/topics/glossary/content/virtual-machine (visited on 03/18/2021).

[6] InsideHPC Report. *Charliecloud ~ Unprivileged Containers for User-Defined Software Stacks*. Apr. 15, 2018. URL: https://www.youtube.com/watch?v=ESsZgcaP-ZQ (visited on 03/05/2021).

[7] *Empowering App Development for Developers — Docker*. URL: https://www.docker.com/ (visited on 03/17/2021).

[8] *Docker github*. GitHub. URL: https://github.com/docker (visited on 03/17/2021).

[9] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. *Inceptionism: Going Deeper into Neural Networks*. Google AI Blog. July 13, 2015. URL: http://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html (visited on 04/03/2021).

[10] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1, 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (visited on 04/03/2021).

[11] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. 1. Edition. s.l.: The Nature of Code, 2012. 520 pp. ISBN: 978-0-9859308-0-6.

[12] *8-usernamespace/create-username-dockremap.sh · master · Xavier / Tutoriels docker*. URL: https://gitlab.com/xavki/presentations_docker/blob/master/8-usernamespace/create-username-dockremap.sh (visited on 03/29/2021).

[13] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLOS ONE* 12.5 (May 11, 2017). Publisher: Public Library of Science, e0177459. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177459 (visited on 03/15/2021).

[14] *NVIDIA/enroot*. original-date: 2018-10-17T19:08:58Z. Feb. 25, 2021. URL: https://github.com/NVIDIA/enroot (visited on 03/04/2021).

[15] Reid Priedhorsky and Tim Randles. "Charliecloud: unprivileged containers for user-defined software stacks in HPC". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17: The International Conference for High Performance Computing, Networking, Storage and Analysis. Denver Colorado: ACM, Nov. 12, 2017, pp. 1–10. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126925. URL: https://dl.acm.org/doi/10.1145/3126908.3126925 (visited on 03/15/2021).

[16] *hpc/charliecloud*. original-date: 2015-06-16T17:04:19Z. Mar. 16, 2021. URL: https://github.com/hpc/charliecloud (visited on 03/23/2021).

[17] Lisa Gerhardt et al. "Shifter: Containers for HPC". In: *Journal of Physics: Conference Series* 898.8 (Oct. 1, 2017). Publisher: IOP Publishing, p. 082021. ISSN: 1742-6596. DOI: 10.1088/1742-6596/898/8/082021. URL: https://iopscience.iop.org/article/10.1088/1742-6596/898/8/082021/meta (visited on 03/15/2021).

[18] *containers/podman*. original-date: 2017-11-01T15:01:27Z. Mar. 15, 2021. URL: https://github.com/containers/podman (visited on 03/15/2021).

[19] James Somers. "Whispers of A.I.'s Modular Future". In: *The New Yorker* (Feb. 1, 2023). URL: https://www.newyorker.com/tech/annals-of-technology/whispers-of-ais-modular-future (visited on 10/22/2023).

[20] Rich Sutton. *The Bitter Lesson*. The Bitter Lesson. Mar. 13, 2019. URL: http://www.incompleteideas.net/IncIdeas/BitterLesson.html (visited on 02/04/2023).

[21] Alec Radford et al. "Robust Speech Recognition via Large-Scale Weak Supervision". In: ().

[22] J. P. Mesirov. "Accessible Reproducible Research". In: *Science* 327.5964 (Jan. 22, 2010), pp. 415–416. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1179653. URL: https://www.sciencemag.org/lookup/doi/10.1126/science.1179653 (visited on 04/04/2021).

[23] Mark D. Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* 3.1 (Dec. 2016), p. 160018. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18. URL: http://www.nature.com/articles/sdata201618 (visited on 12/10/2020).

[24] Gordon E Moore. "Cramming more components onto integrated circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[25] NVIDIA. "Optimized Containers from NVIDIA GPU Cloud". In: (), p. 18. URL: https://www.nvidia.com/en-us/gpu-cloud/containers/.

[26] Daniel Gruber, Burak Yenier, and Wolfgang Gentzsch. "Kubernetes, UberCloud Containers, and HPC". In: (Sept. 23, 2019), p. 7.

[27] John Tomlinson. "History and impact of the intermodal shipping container". In: *Pratt Institute* (2009).

[28] ANSSI. *Recommandations de sécurité relatives au déploiement de conteneurs Docker*. Security Report 1. Paris: Agence Nationale de la Sécurité des Systèmes d'Information, Sept. 23, 2020, p. 22. URL: https://www.ssi.gouv.fr/guide/recommandations-de-securite-relatives-au-deploiement-de-conteneurs-docker/.