Introduction to Cloud COMPUTING

Giuliano Taffoni- I.N.A.F.

"**Container computing"**

2024 @ Università di Trieste

Outline

Intro to Container Technology

Container demo

Outline of this lecture

1. From VM to Containers
2. Why do we need Containers?
3. What is a Container.
4. Container, Container Images, Container orchestration.
5. Simple Tutorial: Building & Running Containers using Docker
6. Why use Containers?

Virtual Environment

Pros

- Reproducible research
- Explicit dependencies
- Improved engineering collaboration

IN PYTHON YOU CAN MAKE A SIMILAR THING

Cons

- Difficulty setting up your environment
- Not isolation
- Does not always work across different OS

⚠ IT DEPENDS ON THE OS

Virtual Machines

Pros

- Full autonomy
- Very secure
- Lower costs
- Used by all Cloud providers for on demand server instances

Cons

- Uses hardware in local machine
- Not very portable since size of VMs are large
- There is an overhead associated with virtual machines

One solution for multiple challenges…

Install a complex scientific pipeline with a large set of dependencies (…what can goes wrong..)

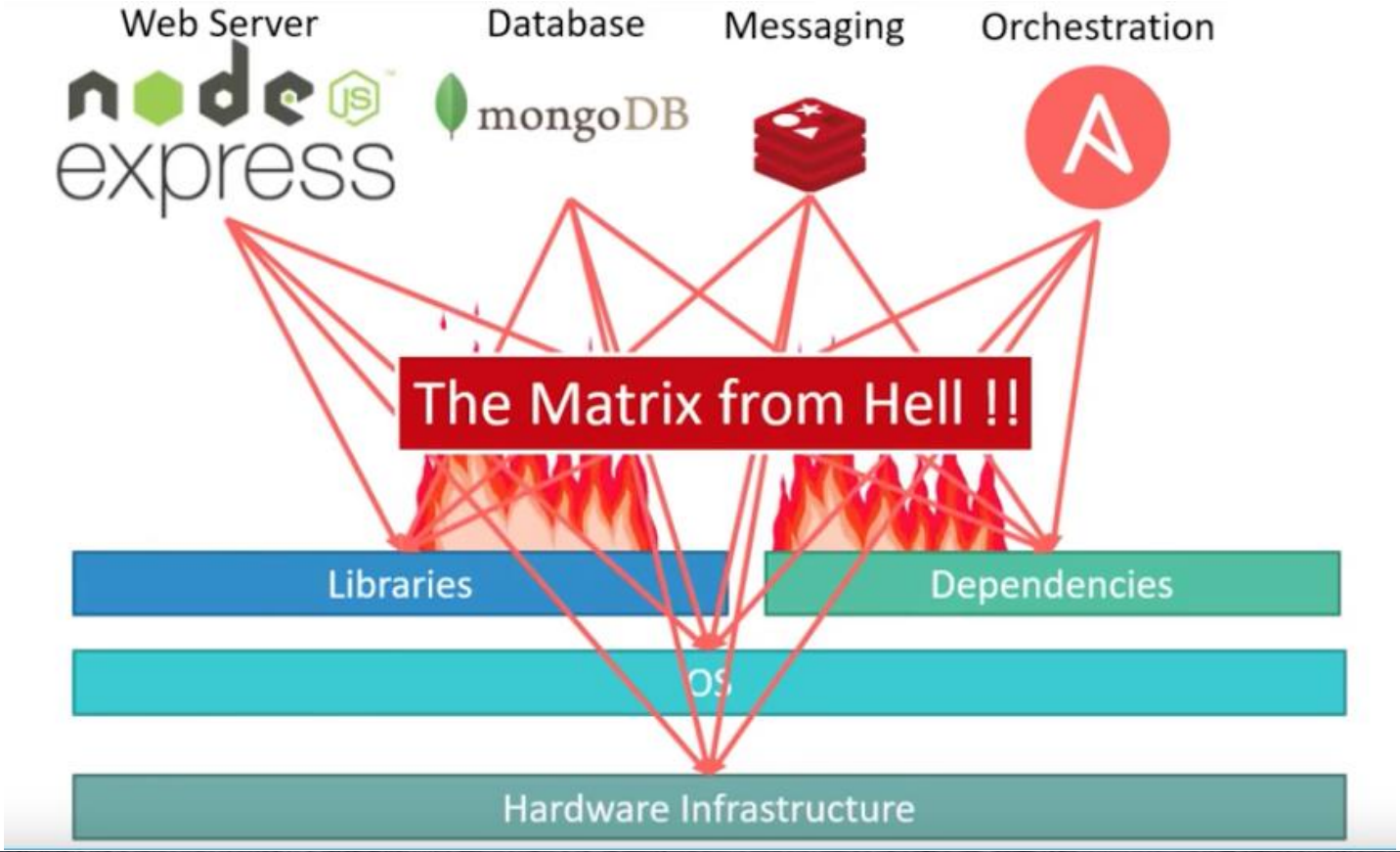
Multiple version of the same software or libraries

Isolated instances of the same application/service

Complex applications with multiple dependences (DBs, Files, webapps, messaging…)

Startups quickly

Dependency Hell Problem



Grasping Containers *Informally*

A container is a standard unit of software that packages up code and all its dependencies in processes isolated from resources, so the application runs quickly and reliably from one computing environment to another.

Containers creates an isolated environment at application level and not at server level.

BUT HOW CAN IT RUN?
LET'S KEEP THE OS SEPARETED (THE KERNEL) AND PUT ALL OTHER THINGS IN A PACKAGE (A CONTAINER). IT'S AN IMMAGE

- ONLY WITH LINUX
- Extremely portable and lightweight;
- Fully packaged software with all dependencies included;
- Can be used for development, training, and deployment;
- Development teams can easily share containers.

What is a container image?

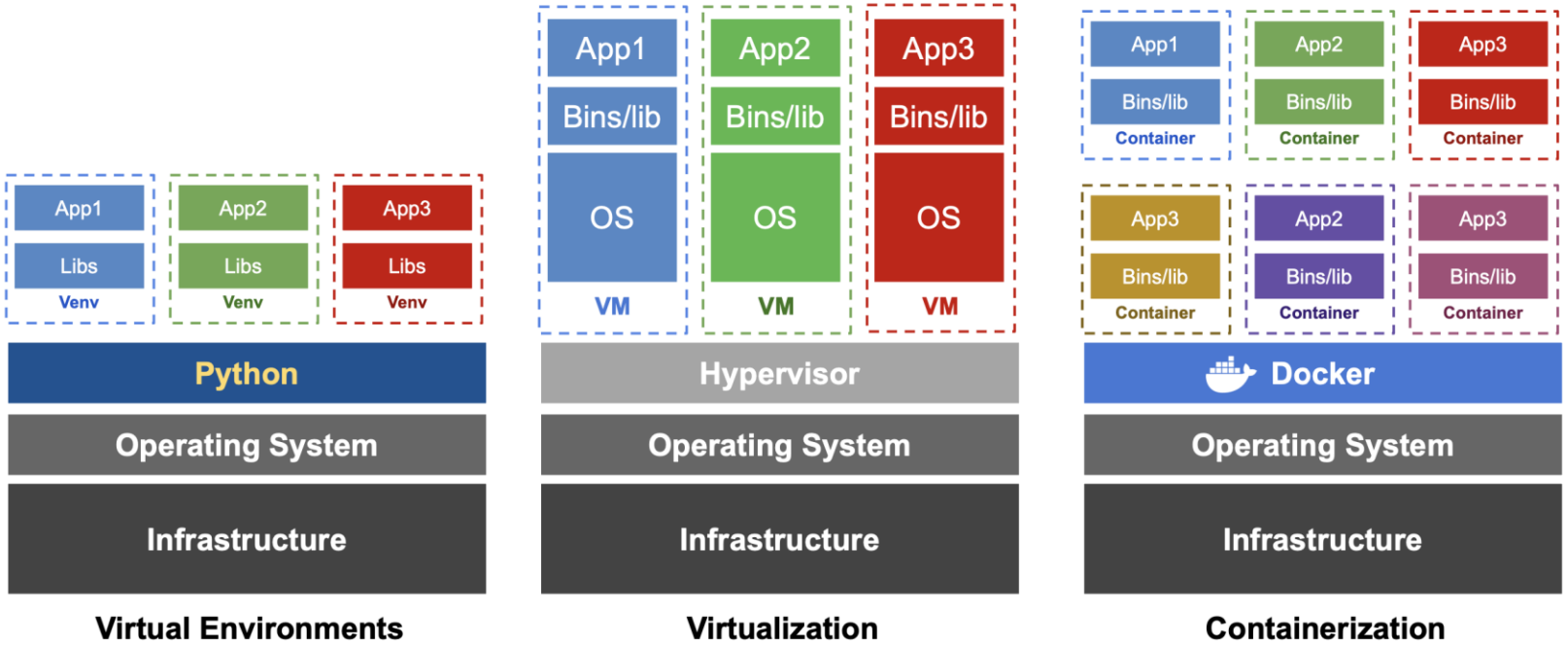
Container images are lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime.

Image is like a recipe; container is like a dish.

Alternatively, you can think of an image as a class and a container is an instance of that class.

VEnv - Virt - Containers



Container vs VMs

★ Virtual Machines can run different Operative Systems (eg. BSD, Windows) and different Architectures (e.g. Arm, SPARK, RISK, PowerPC);

★ Container Engine: abstraction and isolation level between OS and applications environment. Enables and disables containers, manage container life cycle and monitor them;

★ Containers virtualize at the OS level, with multiple containers running atop the OS kernel directly.

★ Containers are far more lightweight: they are “applications” that share the OS kernel, start much faster, and use a fraction of the memory compared to booting an entire OS.

AND THAT'S WHY IT'S ON LINUX

Container Properties

★ Isolation Containers virtualize CPU, memory, storage, and network resources at the OS-level, providing developers with a sandboxed view of the OS logically isolated from other applications. Developers, using containers, are able to create predictable environments isolated from other applications.

★ Productivity enhancement Containers can include software dependencies needed by the application (specific versions of programming language runtimes, software libraries) guaranteed to be consistent no matter where the application is deployed. All this translates to productivity: developers and IT operations teams spend less time debugging and diagnosing differences in environments, and more time shipping new functionality for users.

Container Properties

★ Deployment simplicity containers allow your application as a whole to be packaged, abstracting away the operating system, the machine, and even the code itself, so development and deployment are easier because containers are able to run virtually anywhere (Linux, Windows, and Mac operating systems; virtual machines or bare metal; developer's machine or data centers on-premises; public cloud).

★ Easy portability Docker image format for containers further helps with portability. Docker V2 image manifest is a specification for container images that allows multi-architecture images and supports content-addressable images

★ Easy Versioning A new container can be packaged for each new application version including all needed dependencies,

Container properties

★ Operational efficiency and reliability Containers are perfect for Service Oriented Architectures/Applications because each service limited to specific resources can be containerized. Separate services can be considered as black boxes.

- This arises efficiency because each container can be healt checked and started/stopped when needed independently from others
- Reliability arises because separation and division of labor allows each service to continue running even if others are failing, keeping the application as a whole more reliable

★ Security Containers add an additional layer of security since the applications aren't running directly on the host operating system. There are security constraint if application running inside containers have root privileges

Docker

★ Docker is a containerization platform that packages your application and all its dependencies together in the form of a docker container to ensure that your application works seamlessly in any environment.

★ Docker Container is a standardized unit which can be created on the fly to deploy a particular application or environment. It could be an Ubuntu container, CentOs container, etc. to full-fill the requirement from an operating system point of view.

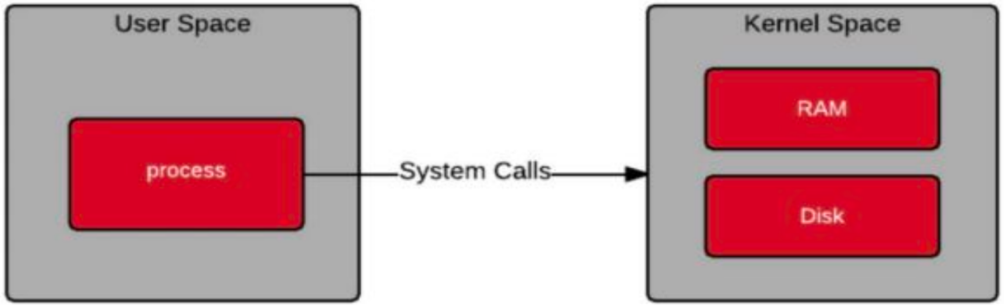
What's behind the scenes…

Containers creates an isolated environment at **application level**: each container is a **process** running including all its children;

Containers underlying (main) technologies:

- Linux NAMESPACEs
- Linux control groups

User space refers to all the code in an operating system that lives outside of the kernel



Process Virtualization

Linux Namespaces

Linux namespaces provide a mechanism for isolating system resources, enabling processes within a namespace to have their own view of the system, such as process IDs,

Namespaces in Linux provide a way to isolate and virtualize system resources, thus enhancing security by preventing processes in one namespace from directly interacting with processes in another namespace.

Namespaces increase security by providing a level of isolation that prevents unintended interactions between processes. This isolation is particularly valuable in containerization and virtualization scenarios, where multiple applications or services share the same host system but must be kept separate for security reasons.

Linux Namespaces

A user namespace has its own set of user IDs and group IDs for assignment to processes. In particular, this means that a process can have root privilege within its user namespace without having it in other user namespaces.

A process ID (PID) namespace assigns a set of PIDs to processes that are independent from the set of PIDs in other namespaces. The first process created in a new namespace has PID 1 and child processes are assigned subsequent PIDs. If a child process is created with its own PID namespace, it has PID 1 in that namespace as well as its PID in the parent process' namespace.

A network namespace has an independent network stack: its own private routing table, set of IP addresses, socket listing, connection tracking table, firewall, and other network-related resources.

Linux Namespaces

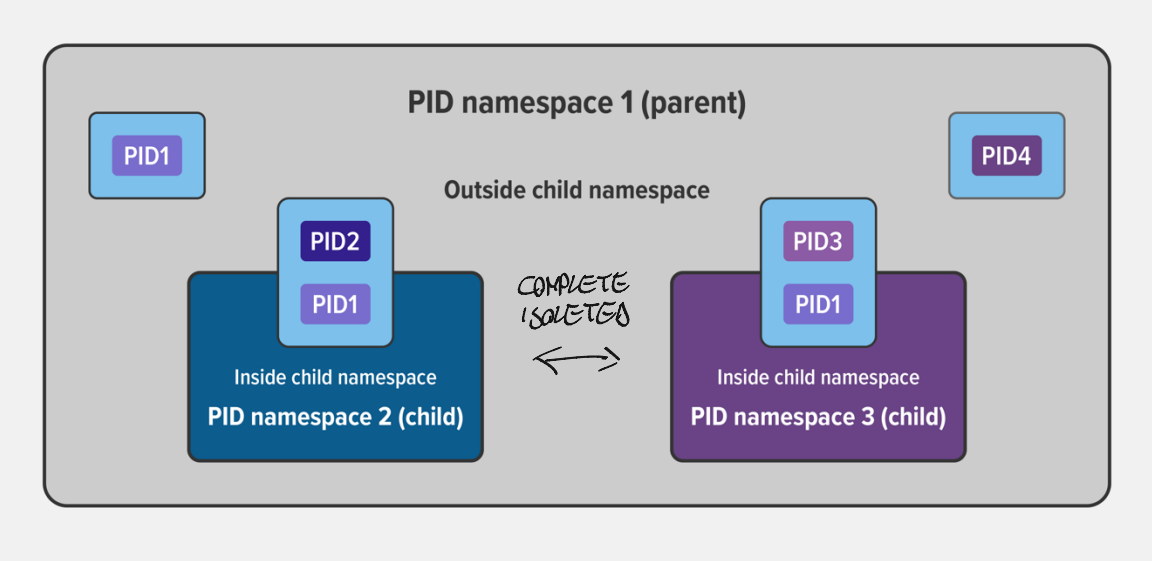
A mount namespace has an independent list of mount points seen by the processes in the namespace. This means that you can mount and unmount filesystems in a mount namespace without affecting the host filesystem.

An interprocess communication (IPC) namespace has its own IPC resources, for example POSIX message queues.

A UNIX Time-Sharing (UTS) namespace allows a single system to appear to have different host and domain names to different processes.

Process Namespace Isolation

Within the parent namespace, there are four processes, named PID1 through PID4. These are normal processes which can all see each other and share resources.



From within a child namespace, the PID1(in PID2) process cannot see anything outside.

Testing the namespaces

unshare - run program in new name namespaces

```
$ sudo ps -ef
$ ps -ef
$ sudo lsns
$ ls
$ unshare -user -pid -map-root-user -mount-proc -fork bash

# ps -ef

$ lsns -output-all
```

Run bash in a new name space as **root**!!!!

Namespaces and containers

Namespaces are one of the technologies that containers are built on, used to enforce segregation of resources. We've shown how to create namespaces manually, but container runtimes like Docker, rkt, and podman make things easier by creating namespaces on your behalf.

Control Groups

A control group (cgroup) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, and so on) of a collection of processes.
Cgroups provide the following features:
Resource limits – You can configure a cgroup to limit how much of a particular resource (memory or CPU, for example) a process can use.

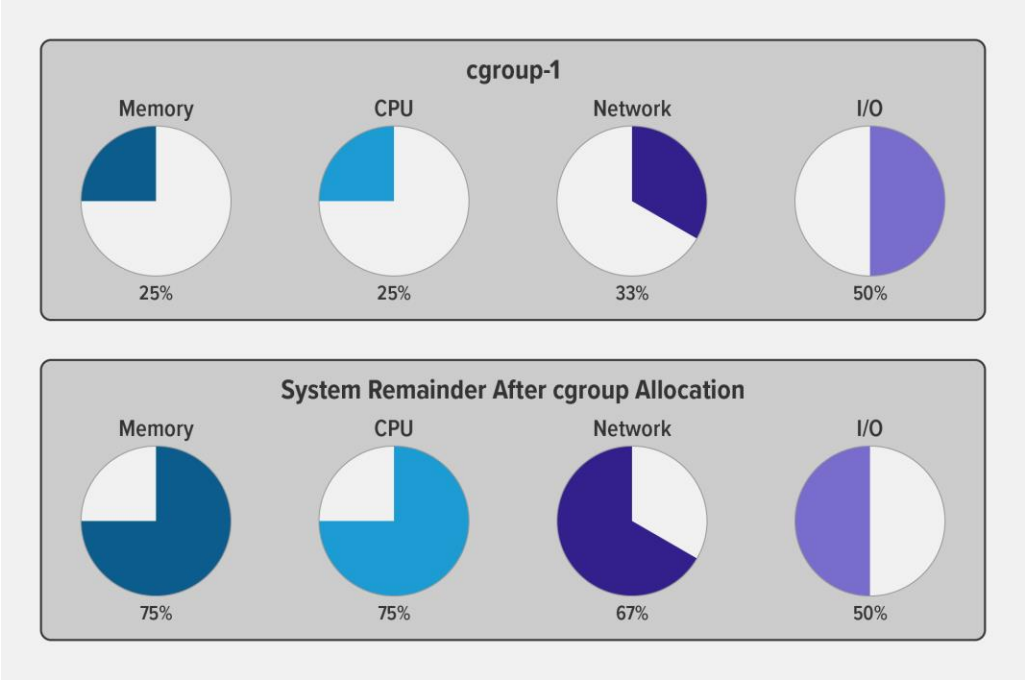
Prioritization – You can control how much of a resource (CPU, disk, or network) a process can use compared to processes in another cgroup when there is resource contention.

Accounting – Resource limits are monitored and reported at the cgroup level.

Control – You can change the status (frozen, stopped, or restarted) of all processes in a cgroup with a single command.

Cgroups

Kernel uses cgroups to control how much of a given key resource (CPU, memory, network, and disk I/O) can be accessed or used by a process or set of processes.



How to implement cgroups

Control groups can be accessed with various tools:

- using directives in systemd unit files to specify limits for services and slices;
- by accessing the cgroup filesystem directly;
- via tools like cgcreate, cgexec and cgclassify (part of the libcgroup and libcgroup-git packages);
- using the "rules engine daemon" to automatically move certain users/groups/commands to groups (/etc/cgrules.conf and cgconfig.service) (part of the libcgroup and libcgroup-git packages);

Some commands to test

```
$ cat /proc/self/cgroup
0::/user.slice/user-1000.slice/session-3.scope
```

The **`systemd-cgtop`** command can be used to see the resource usage:

```
$ systemd-cgtop
Control Group                              Tasks %CPU Memory Input/s Output/s
user.slice                                 540   152,8 3.3G   - -
user.slice/user-1000.slice                 540   152,8 3.3G.  - -
user.slice/u…000.slice/session-1.scope     425   149,5 3.1G   - -
```

Some commands to test

One of the powers of cgroups is that you can create "ad-hoc" groups on the fly.

```
# cgcreate -a user -t user -g memory,cpu:groupname
```

```
root@template:~# cgcreate -a user01 -t user01 -g memory,cpu:mytest
root@template:~# ls /sys/fs/cgroup/mytest/
cgroup.controllers      cgroup.stat             cpu.pressure           cpu.uclamp.max   io.stat             memory.max           memory.swap.events
cgroup.events           cgroup.subtree_control  cpuset.cpus            cpu.uclamp.min   io.weight           memory.min           memory.swap.high
cgroup.freeze           cgroup.threads          cpuset.cpus.effective  cpu.weight       memory.current      memory.numa_stat     memory.swap.max
cgroup.kill             cgroup.type             cpuset.cpus.partition  cpu.weight.nice  memory.events       memory.oom.group     pids.current
cgroup.max.depth        cpu.idle                cpuset.mems            io.max           memory.events.local memory.pressure      pids.events
cgroup.max.descendants  cpu.max                 cpuset.mems.effective  io.pressure      memory.high         memory.stat          pids.max
cgroup.procs            cpu.max.burst           cpu.stat               io.prio.class    memory.low          memory.swap.current
```

Cgroups are hierarchical, so you can create as many subgroups as you like. If a normal user wants to run a bash shell under a new subgroup called foo:

```
$ cgcreate -g memory,cpu:groupname/foo
$ cgexec -g memory,cpu:groupname/foo bash
```

To limit the memory usage of all processes in this group to 10 MB,

```
$ echo 10000000 > /sys/fs/cgroup/memory/mytest/foo/memory.limit_in_bytes
```

Containers and resources

Control groups can be implemented directly by container engines

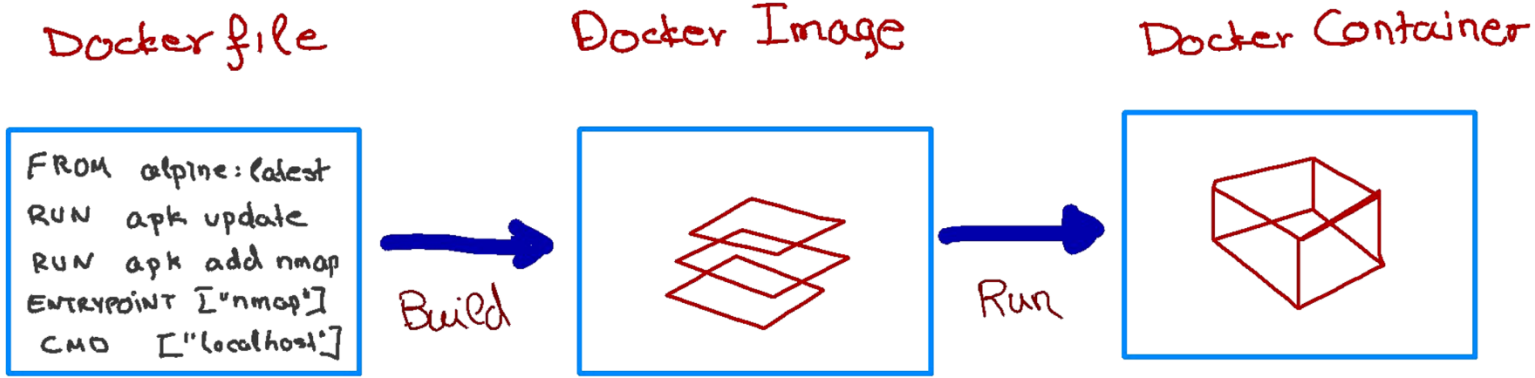
```
$ docker run --cpus 0.5 -d nginx
```

```
$ docker run -d --name new-container --memory=256M
ubuntu sleep infinity
```

What am I doing?

Containers and Images

We use a simple text file, the Dockerfile, to build the Docker Image,



Docker File

```
FROM ubuntu:18.04
RUN apt update
RUN apt upgrade
ENTRYPOINT [“mytask.sh”]
PULL ./FILE.py
CMD python FILE.py
```

FROM: This instruction in the Dockerfile tells the daemon, which base image to use while creating our new Docker image. In the example here, we are using a very minimal OS image called alpine (just 5 MB of size). You can also replace it with Ubuntu, Fedora, Debian or any other OS image.

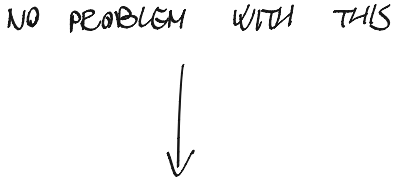
RUN: This command instructs the Docker daemon to run the given commands as it is while creating the image. A Dockerfile can have multiple RUN commands, each of these RUN commands create a new layer in the image.

ENTRYPOINT: The ENTRYPOINT instruction is used when you would like your container to run the same executable every time. Usually, ENTRYPOINT is used in scenarios where you want the container to behave exclusively as if it were the executable it's wrapping.

CMD: The CMD sets default commands and/or parameters when a docker container runs. CMD can be overwritten from the command line via the docker run command.

Multiple containers from same image

FROM ubuntu:18.04
RUN apt update
RUN apt upgrade
ENTRYPOINT [“/bin/echo”, “Hello”]
CMD [”World”]



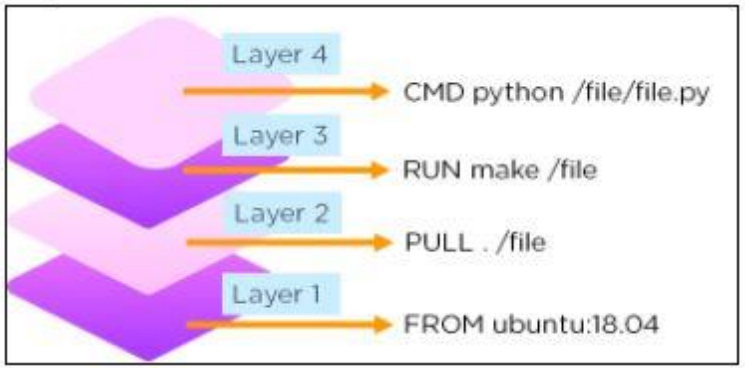

```
> docker build -t hello_world_cmd:first -f Dockerfile_cmd .

> docker run -it hello_world_cmd:first
> Hello world
> docker run -it hello_world_cmd:first Pavlos
> Hello Pavlos
```

Doker Images

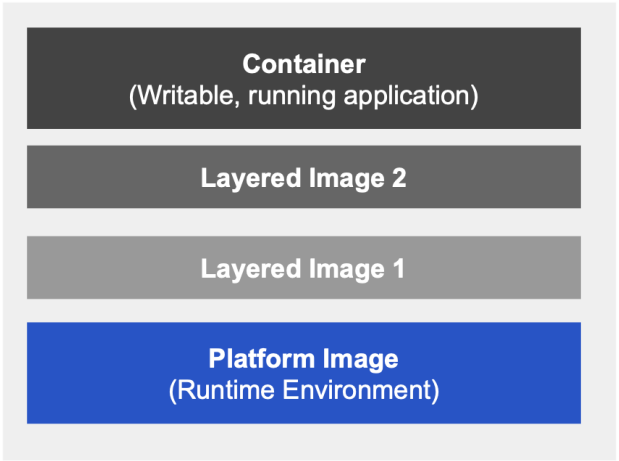
Docker Images are based on layers

IF YOU GOT SOMETHING LIKE THE SLIDE BEFORE, IT'S BECAUSE THEY ARE STARTING ONE AT THE TIME

FROM ubuntu:18.04
PULL. /file
RUN make /file
CMD python /file/file.py



Docker images



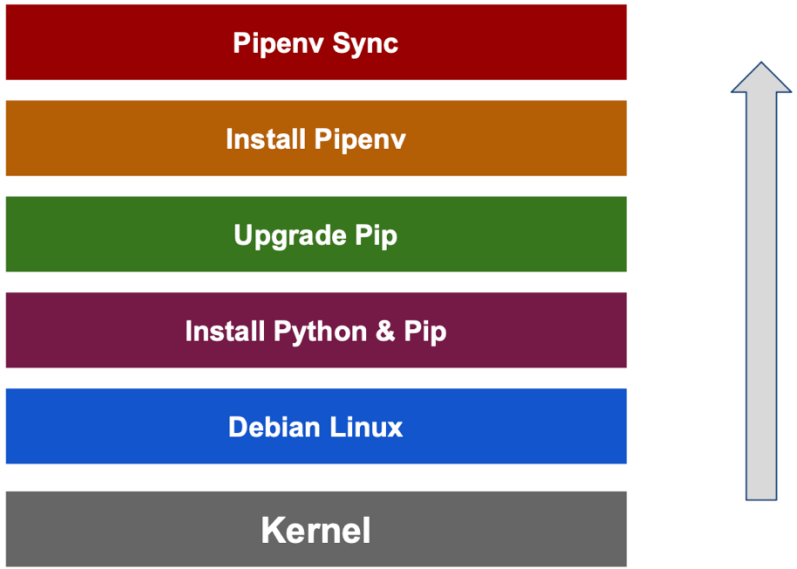
A application sandbox
- Each container is based on an image that holds necessary config data
- When you launch a container, a writable layer is added on top of the image

A static snapshot of the container configuration
- Layer images are read-only
- Each image depends on one or more parent images

An Image that has no parent
- Platform images define the runtime environment, packages and utilities necessary for containerized application to run

Layer example

Docker layers for a container running debian and a python environment using Pipenv



$umming up formal definitions

A container is **a standard Linux process** typically created through a clone() system call instead of fork() or exec(). Also, containers are often isolated further through the use of:cgroups, namespaces, SELinux or AppArmor

A container is the **runtime instantiation** of a Container Image.

Container Image is a package of software, it is used as mountpoint for containers. Usually, it bundles multiple container Image Layers as well as metadata which provides extra information about the layers.

The **container host** is the system that runs the containerized processes, often simply called containers.

$umming up formal definitions

A container engine is a piece of software that accepts user requests, including command line options, pulls images, and from the end user's perspective runs the container.
The container runtime is responsible for:

- Consuming the container mount point provided by the Container Engine (can also be a plain directory for testing)
- Consuming the container metadata provided by the Container Engine (can be a also be a manually crafted config.json for testing)
- Communicating with the kernel to start containerized processes (clone system call)
- Setting up cgroups
- Setting up SELinux Policy
- Setting up App Armor rules

that’s all, have fun