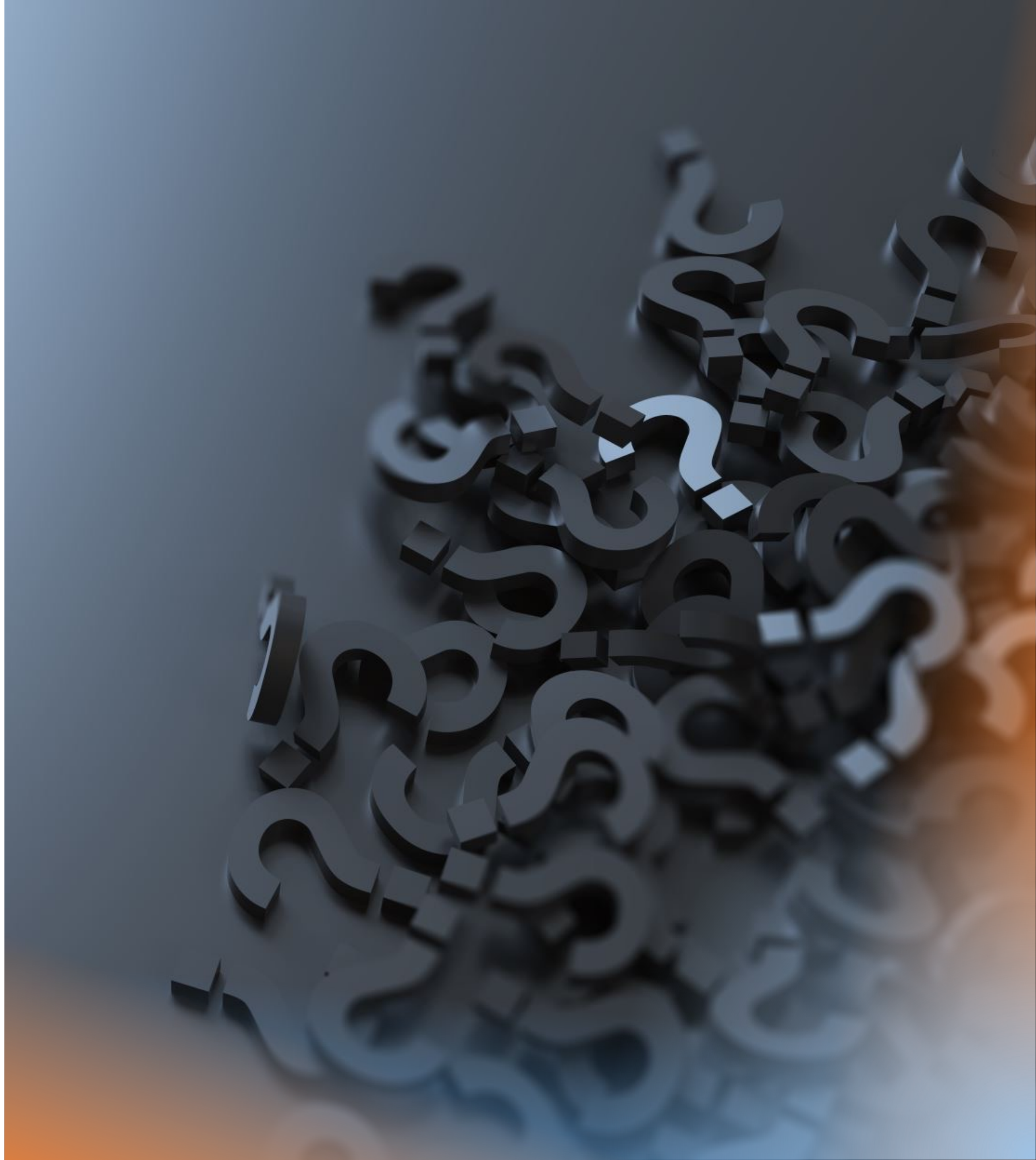
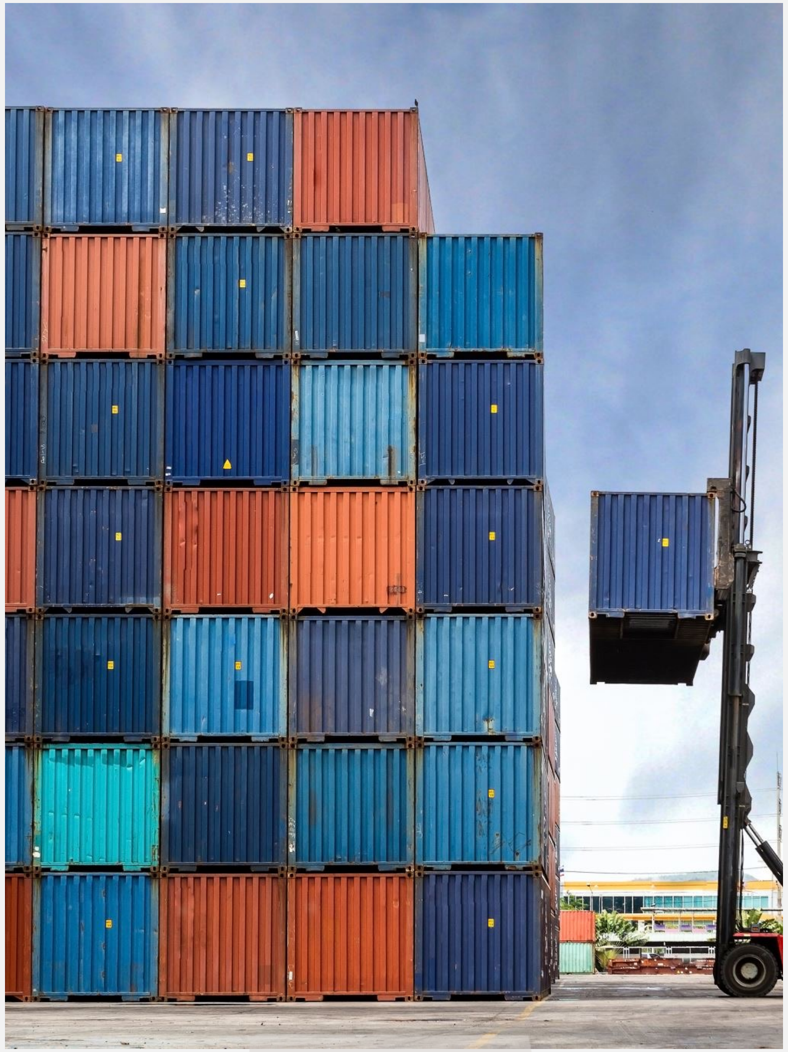
Container Primer

Giuliano Taffoni

The Deal

- We will use Docker as reference, but the concepts are 100% engine-agnostic.
- Always interrupt if you have question, doubts, something not clear, curiosities. Let’s try to keep it interactive!
- Over the talk, think about a concrete use case close to your work. We can discuss a few at the end.





Outline

- Why do we need a container technology in science?
- What is a container?
- How does a container work?
- Container VS Virtual machines
- Images, containers, volumes and networking
- Container engines: docker, podman, singularity…

The dependency hell problem

Mike is a **scientists** that wants to install a new software.

Mike cannot find a precompiled version that works with his OS and/or libraries.

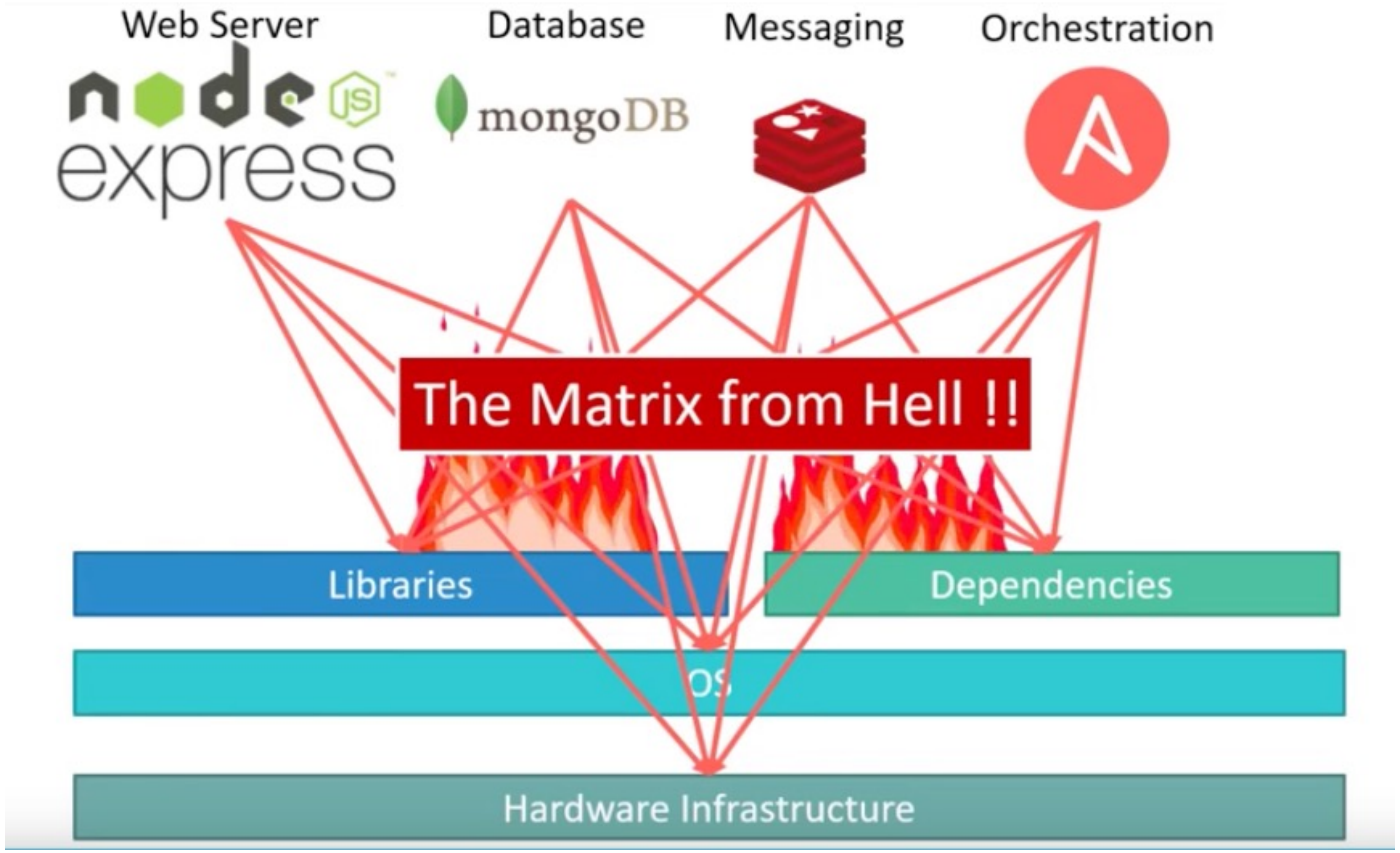
Mike ask/Google for help and get some basic instructions - like “compile it”.

Mike starts downloading all the development environment, and soon realizes that he needs to upgrade (or downgrade!) some parts of his main Operating Systems.

During this process, something goes wrong.

Mikes spends an afternoon fixing his own OS, and all the next day in trying to compile the software. Which at the end turns out not to do what he wanted.

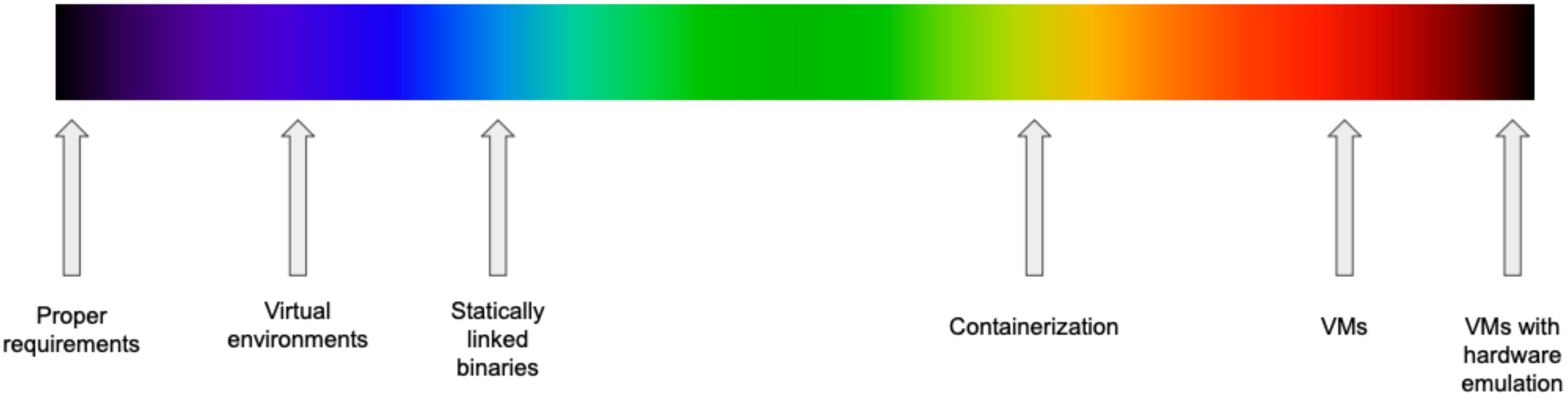
The dependency hell problem



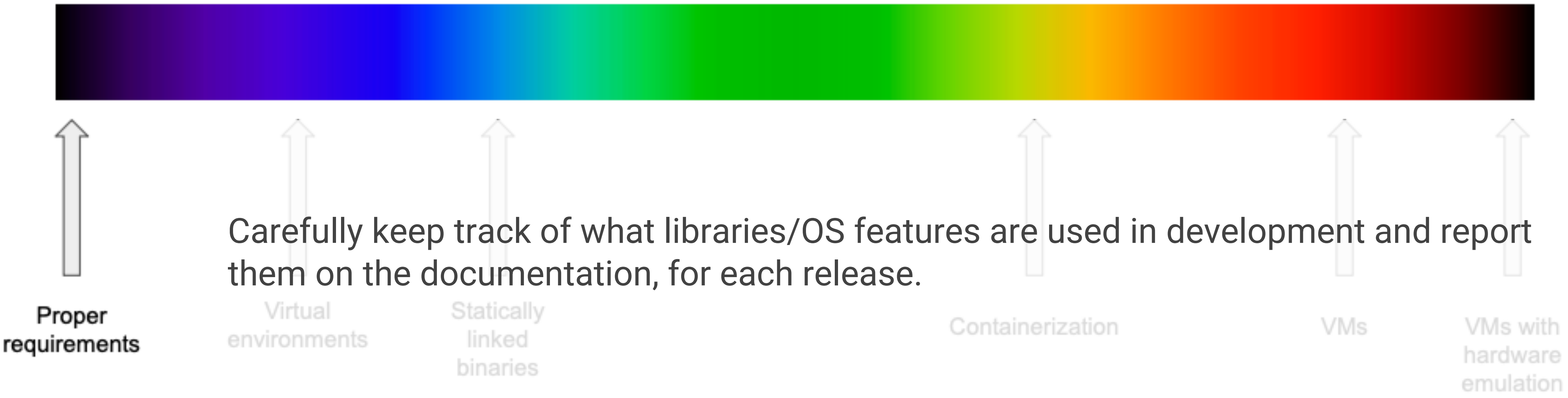
One solution for multiple challenges

- 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…)

Solution spectrum

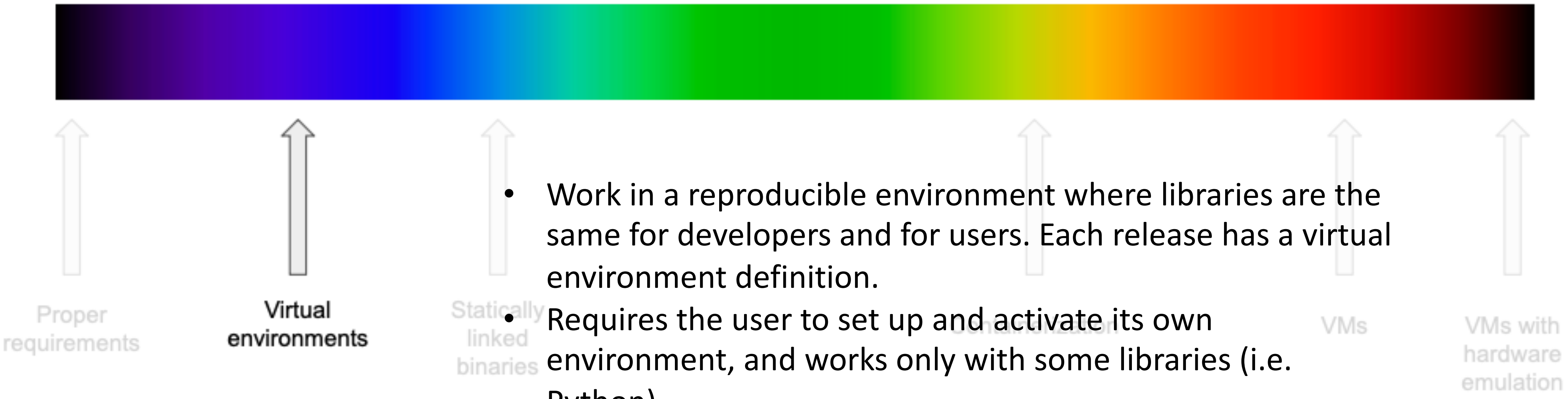


Solution spectrum: proper requirements



Carefully keep track of what libraries/OS features are used in development and report them on the documentation, for each release.

Solution spectrum: virtual environments



- Work in a reproducible environment where libraries are the same for developers and for users. Each release has a virtual environment definition.
- Requires the user to set up and activate its own environment, and works only with some libraries (i.e. Python),
- Not a comprehensive solution and prone to human error

Solution spectrum: virtual environments

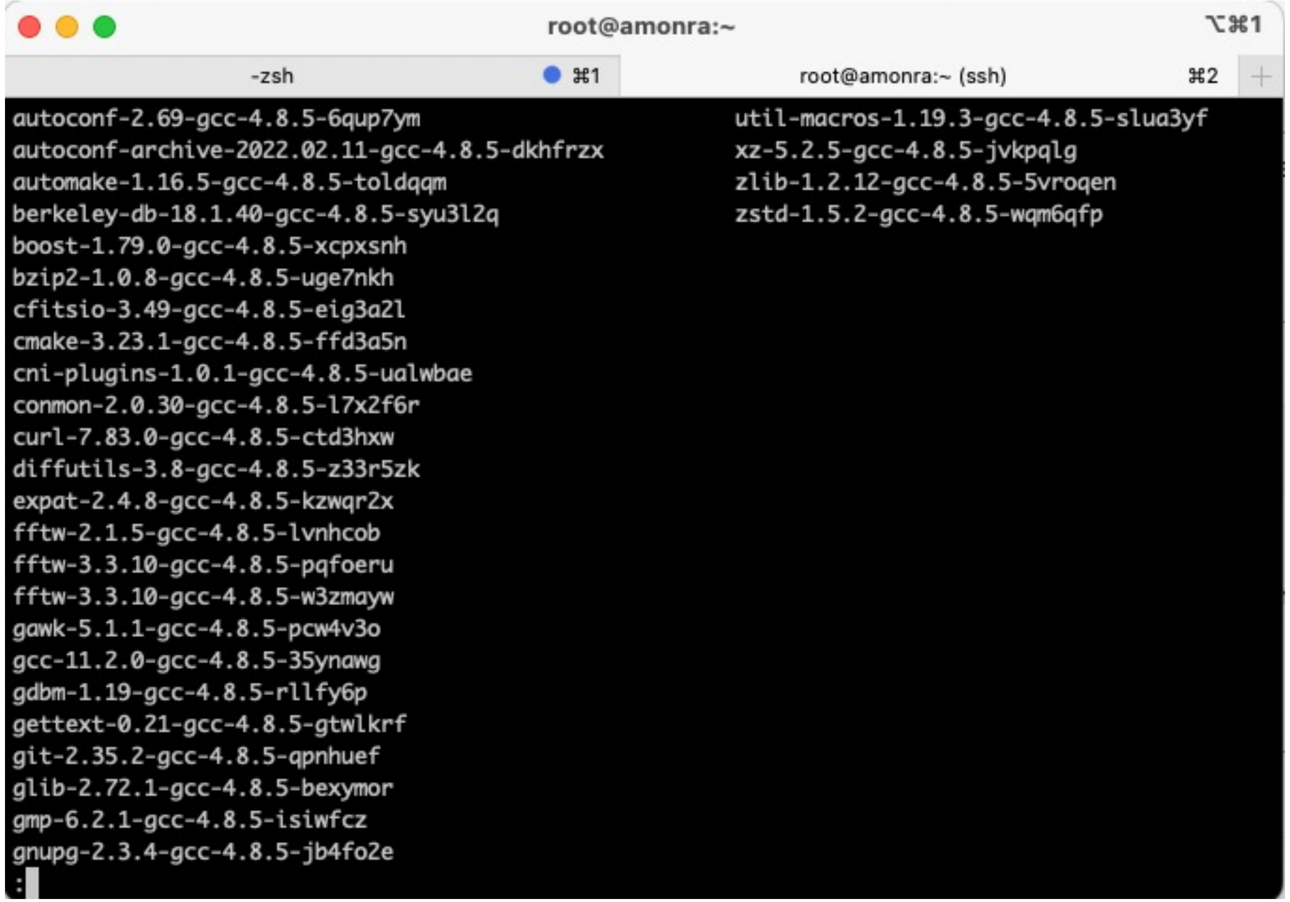
- Python programs often use modules and packages outside of the standard library.
- Python applications require specific versions of a library.
- A single installation can't meet the needs of all applications.

“Virtual Environments: a self-contained directory with a Python installation for a specific Python version and additional packages.”

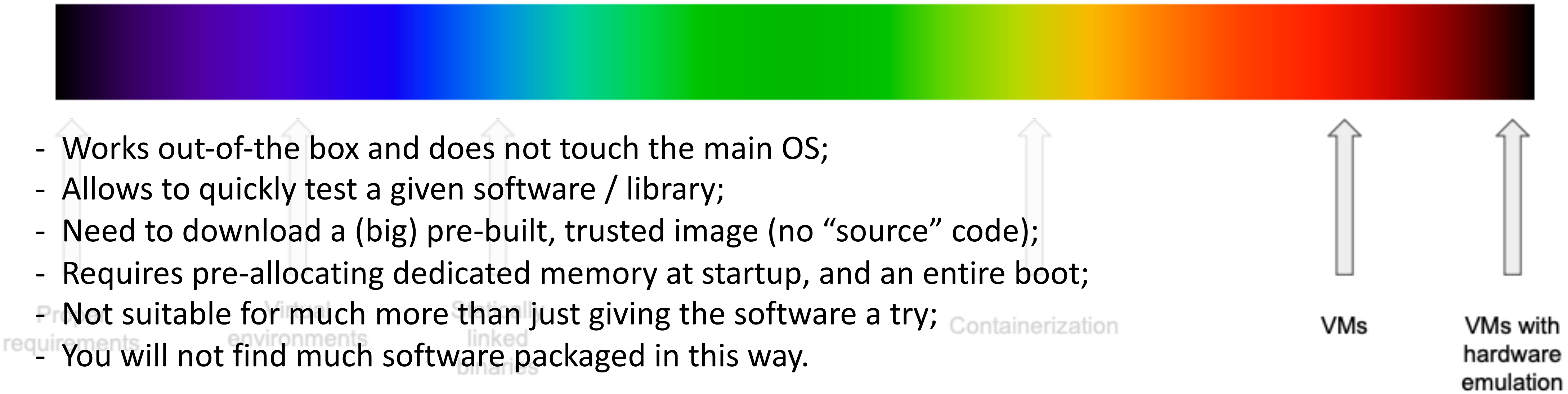
Solution spectrum: virtual environments

Environment Modules for Clusters (HPC/HTC)

- A tool that simplify shell initialization and lets users easily modify their environment during the session with modulefiles.
- Provide simultaneous versions of the same software without collisions, as each module is housed entirely in its own subfolder structure.



Solution spectrum: virtual machine

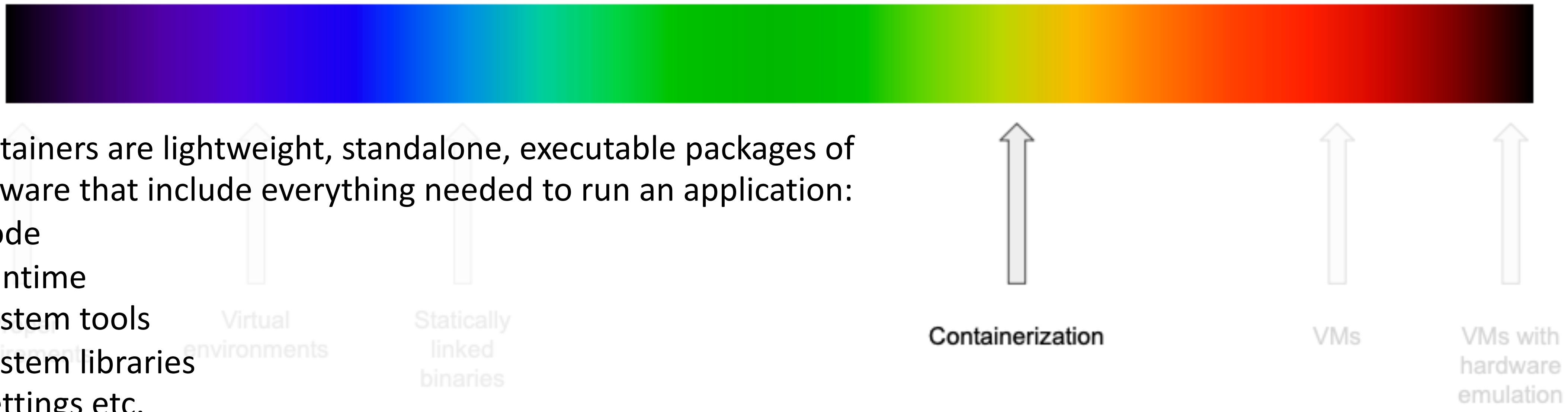


- Works out-of-the box and does not touch the main OS;
- Allows to quickly test a given software / library;
- Need to download a (big) pre-built, trusted image (no “source” code);
- Requires pre-allocating dedicated memory at startup, and an entire boot;
- Not suitable for much more than just giving the software a try;
- You will not find much software packaged in this way.

Virtual machines with hardware emulation... a bit of over-engineering.

... but we are on the right path. We want this kind of insulation!

Solution spectrum: Containers

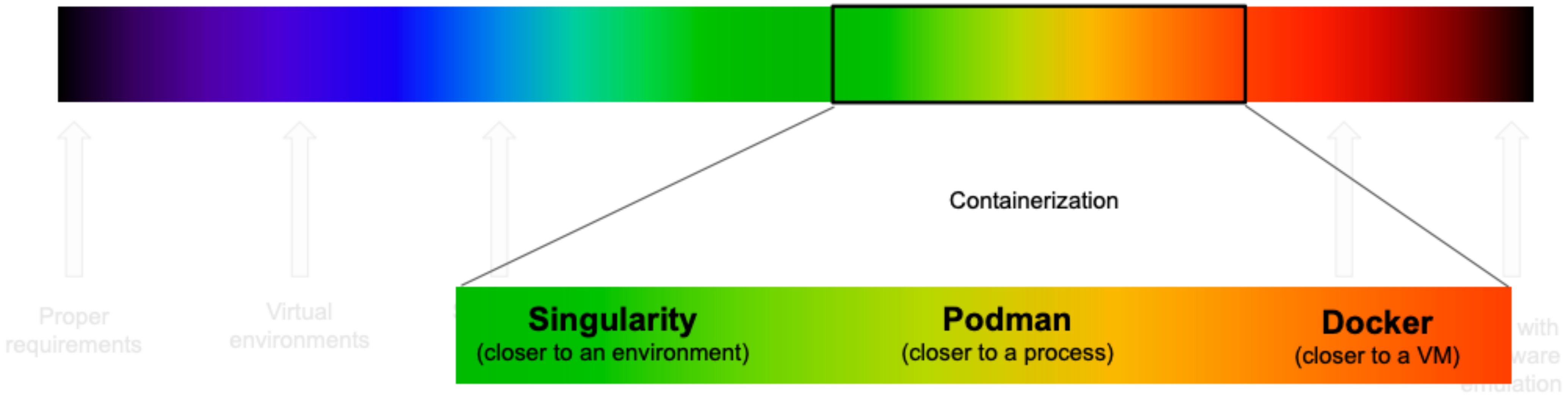


Containers are lightweight, standalone, executable packages of software that include everything needed to run an application:

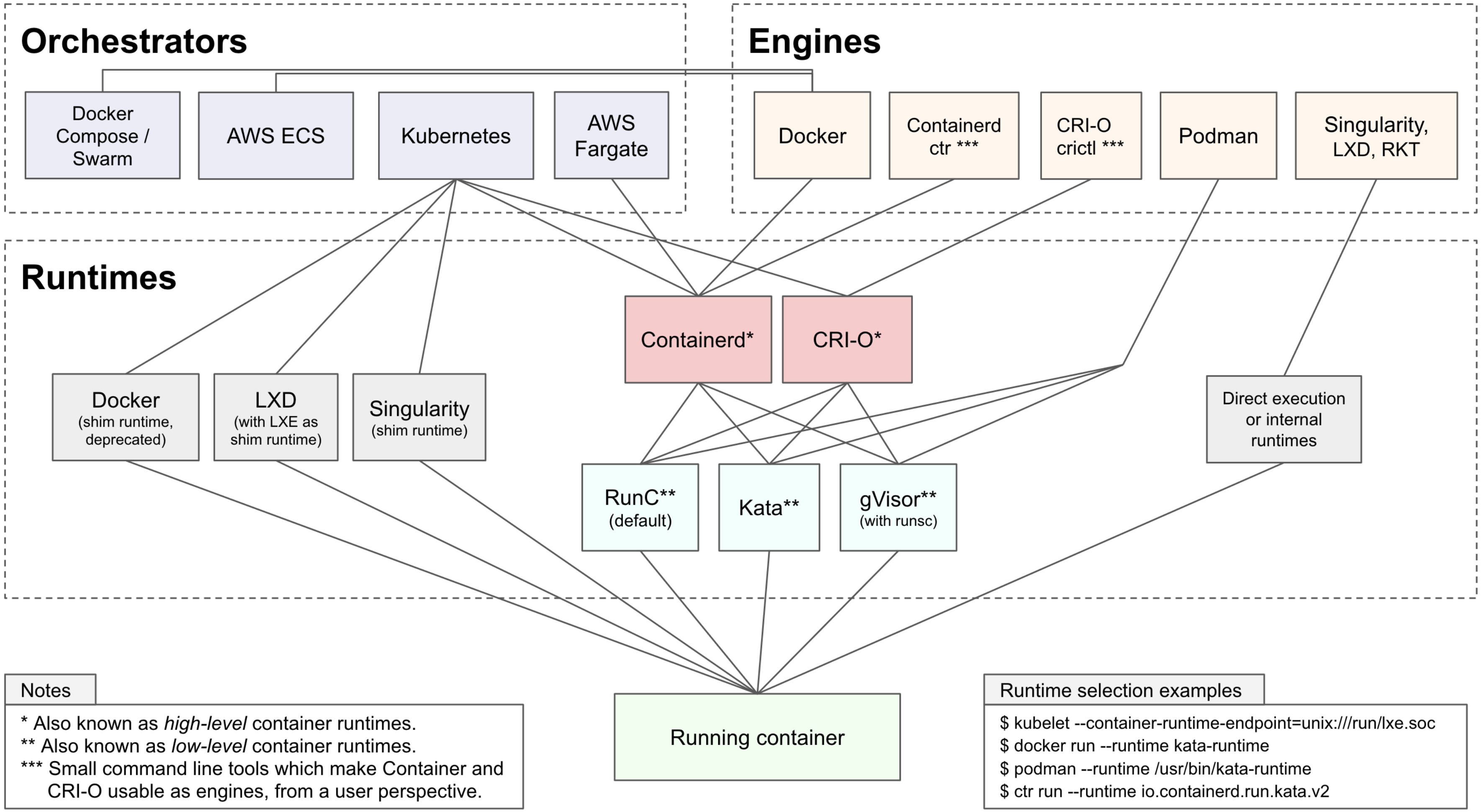
- code
- runtime
- system tools
- system libraries
- settings etc.

Containers allow to reliably move and distribute software from one computing environment to another, without the burden of VMs.

Solution spectrum: Containers



Conteiner ecosystem



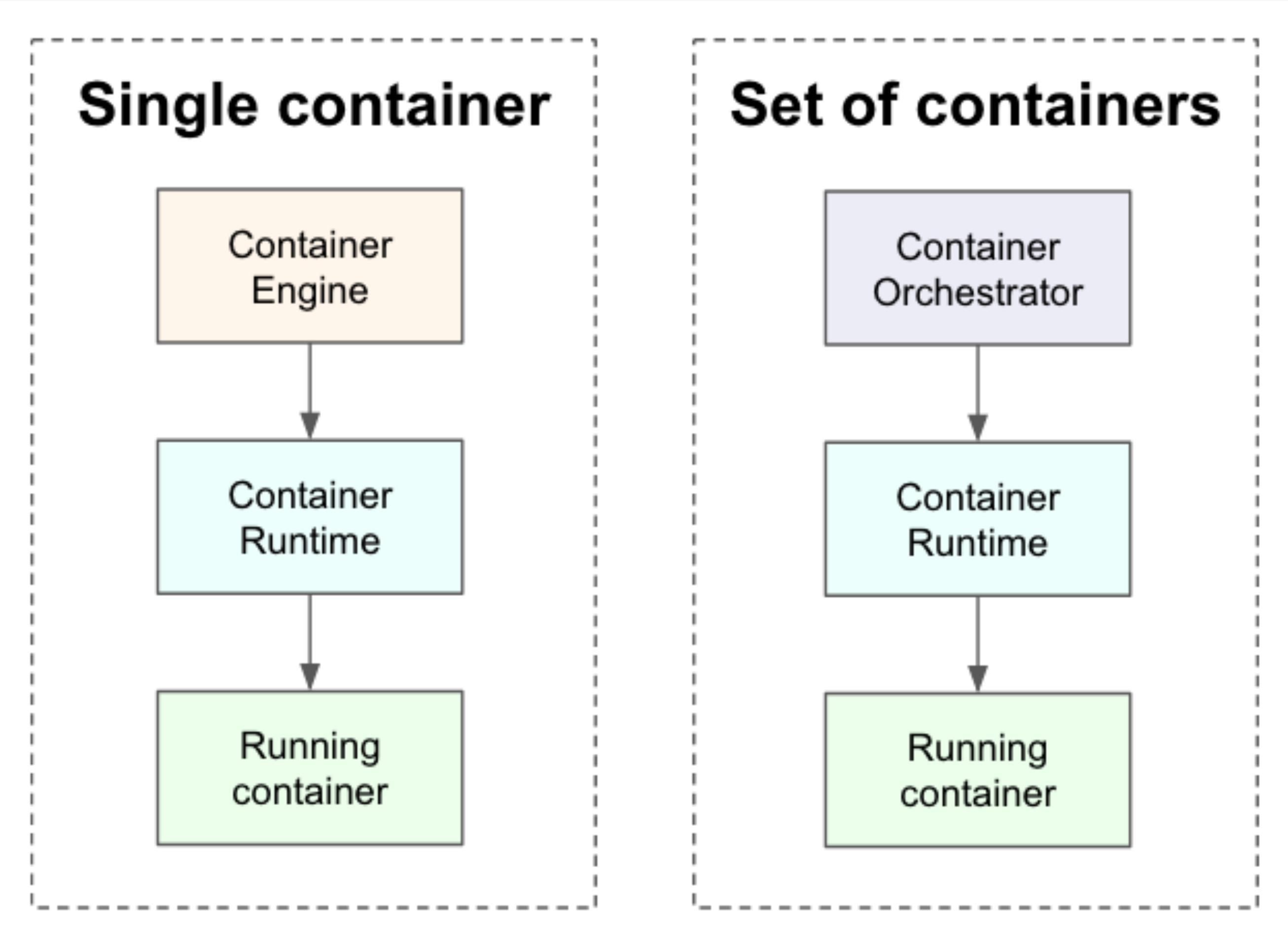
https://sarusso.github.io/blog/container-engines-runtimes-orchestrators.html

Some 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

A **container runtime** is a software component which is in charge of managing the container lifecycle: configuring its environment, running it, stopping it, and so on.

A **container orchestrator** is a software in charge of managing set of containers across different computing resources,

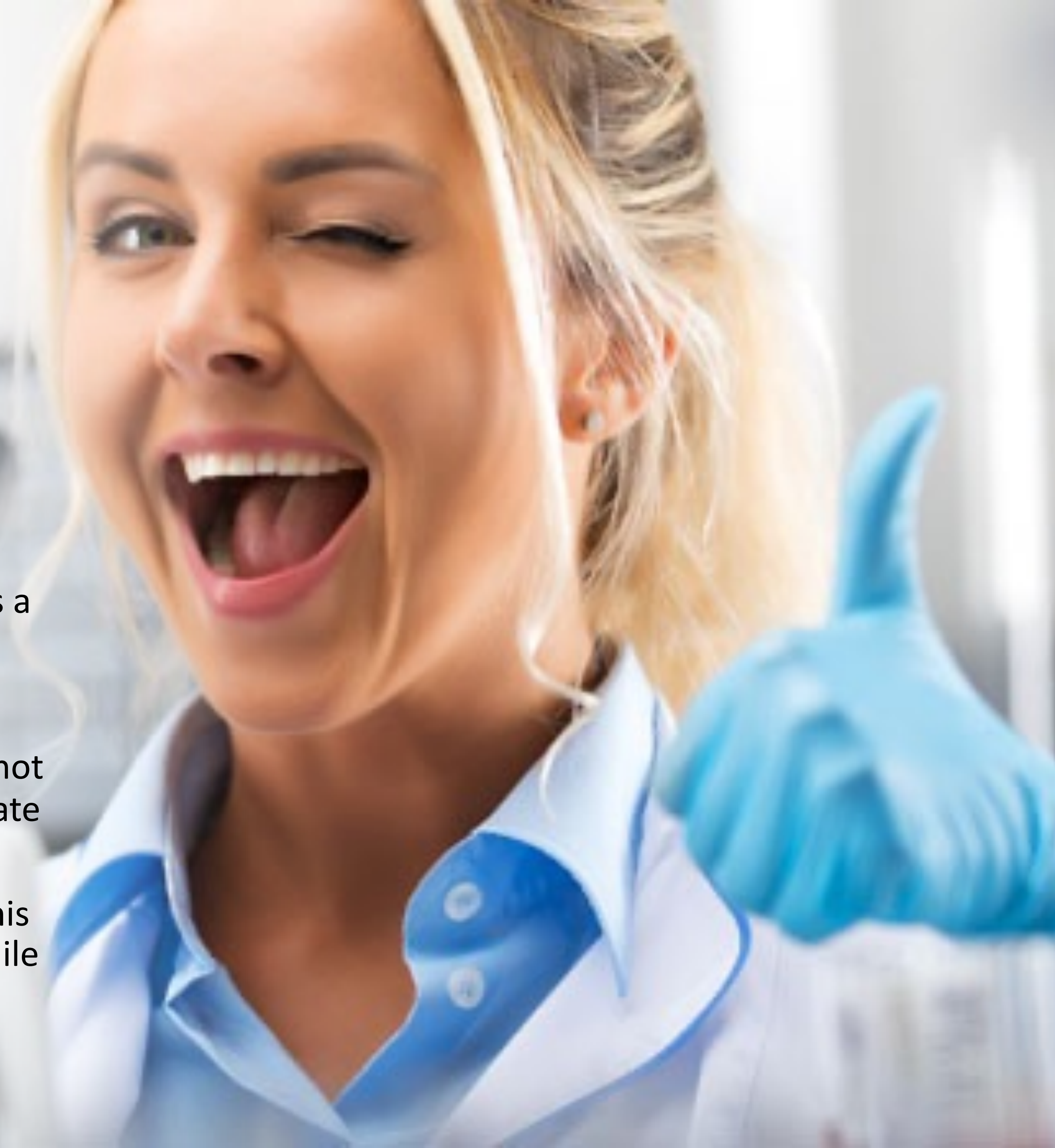


Container enviroments

- If you are running ***single containers***, you will interact with a **container engine**,
- If you are running ***set of containers***, you will then use a **container orchestrator**.

Container for science…but not only

- Jane wants to install a new software.
- Jane cannot find a precompiled version that works with his OS and/or libraries.
- Jane ask/Google for help and finds out that there is a container for it.
- Jane pulls the container and runs it.
- Jane immediately discovers that the software is/is not suitable for his research and finds a more appropriate one (as a container, of course!)
- Jane spends the afternoon writing conclusions on his very important research using his new software while enjoying a hot cup of latte.

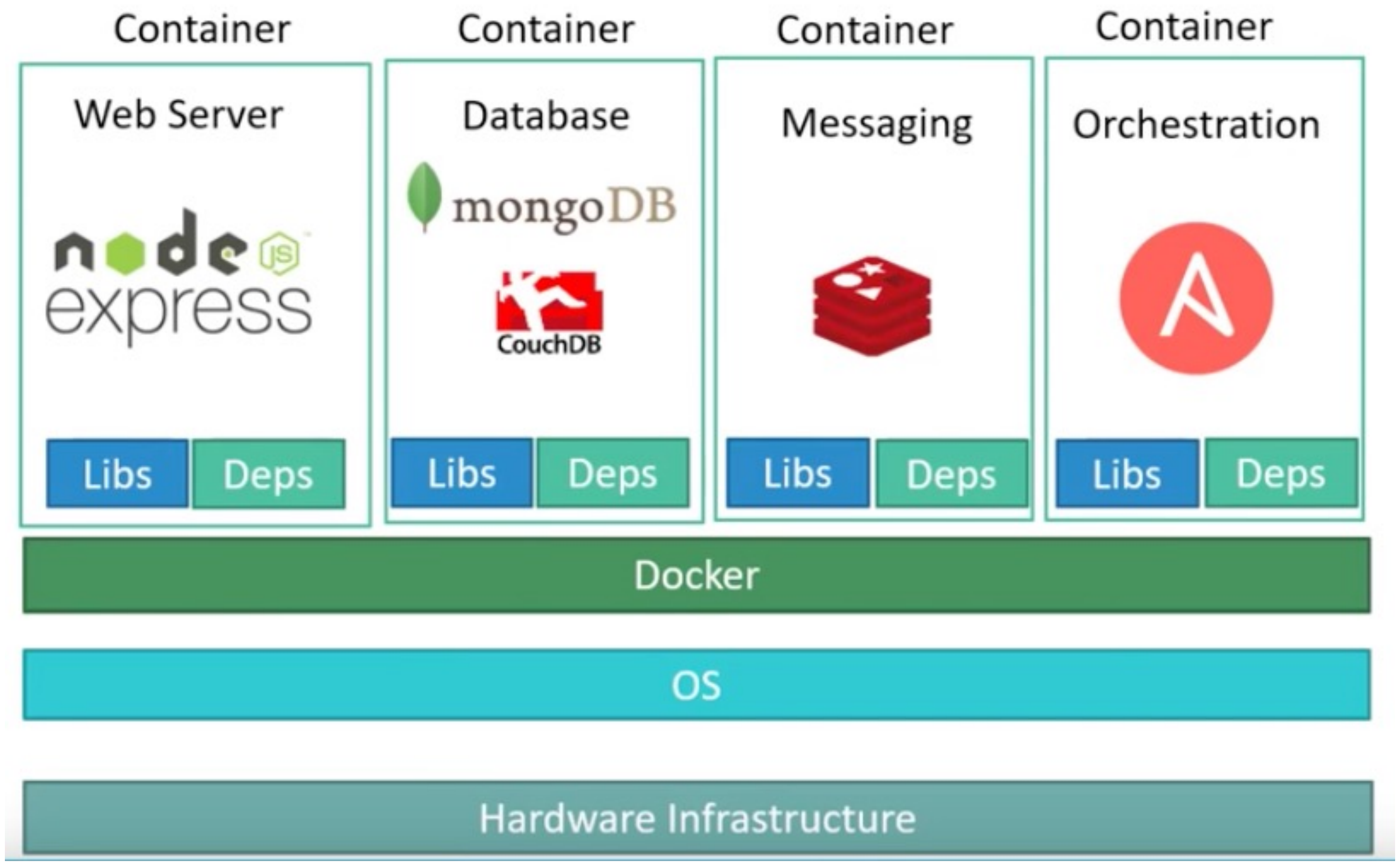


What is a container?

- A container is a standard unit of software that packages up code and all its dependencies.
- Containers creates portable isolated environments at application level and not at server level.
- Insulate a single process from your Operating System, and to:
  - Let it live in its own space, including its own network;
  - Let it have its own File System with its own libraries;
  - Allow to natively access hardware without virtualization;
  - Avoid booting an entire Virtual machine and to pre-allocate dedicated memory.

You might think about them as Virtual Machines in first approximation → but keep in mind that they are two completely different beasts.

Dependency Hell

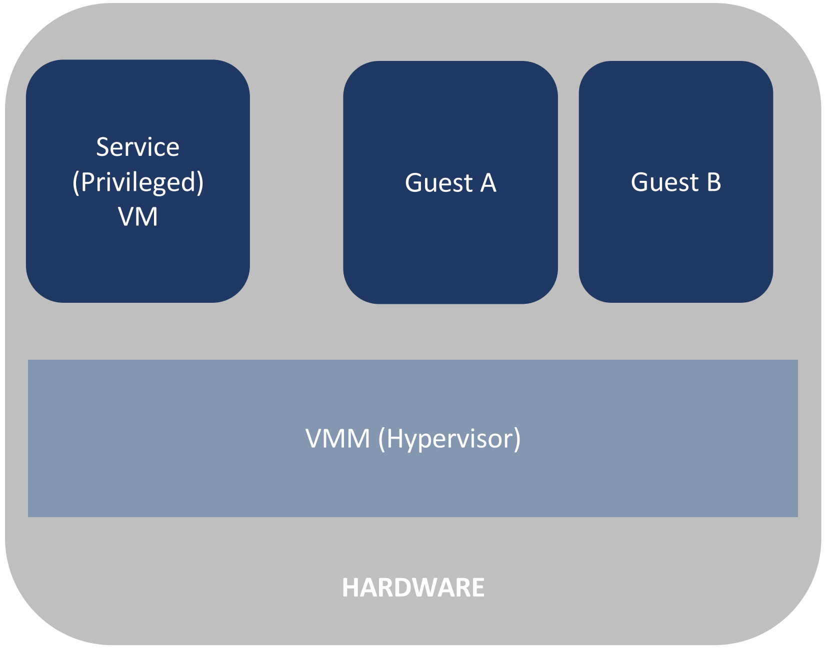


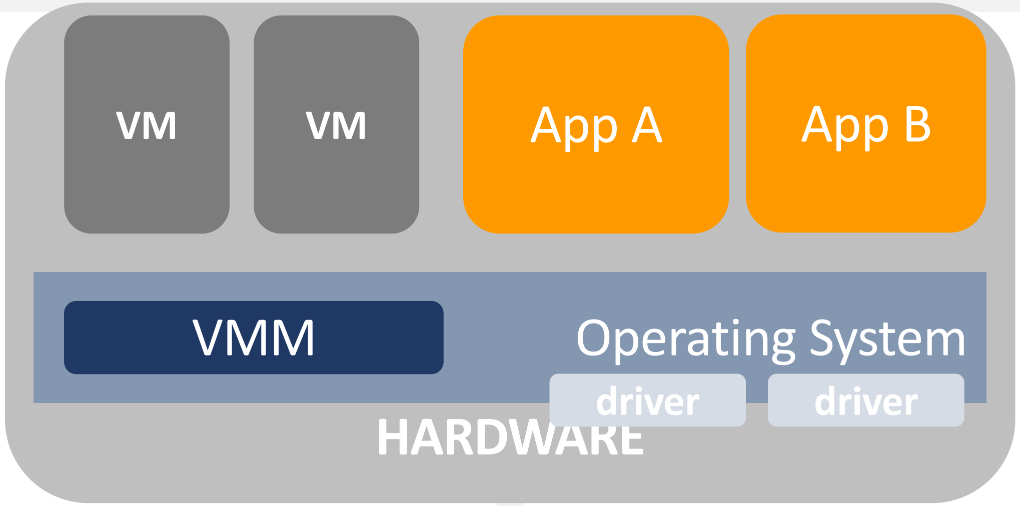
Virtualization primer

Virtualization, is the ability to simulate a hardware platform, such as a server, storage device or network resource, in software. All the functionality is separated (**abstracted**) from the hardware and simulated as a “virtual instance” with the ability to operate just like the hardware solution. A single hardware platform can be used to support multiple virtual devices or machines, which are easy to spin up or down as needed.

Virtual Machine is the software simulation of a computer. It is able to run an Operating Systems and applications interacting with the virtualized abstracted resources, not with the physical resources, of the actual host computer.

Hypervisor (or Virtual Machine Monitor) is a software tool installed on the physical host system to provide the thin software layer of abstraction that decouples the OS from the physical bare-metal. It allows to split a computer in different separate environment, the Virtual Machines, distributing them the computer resources



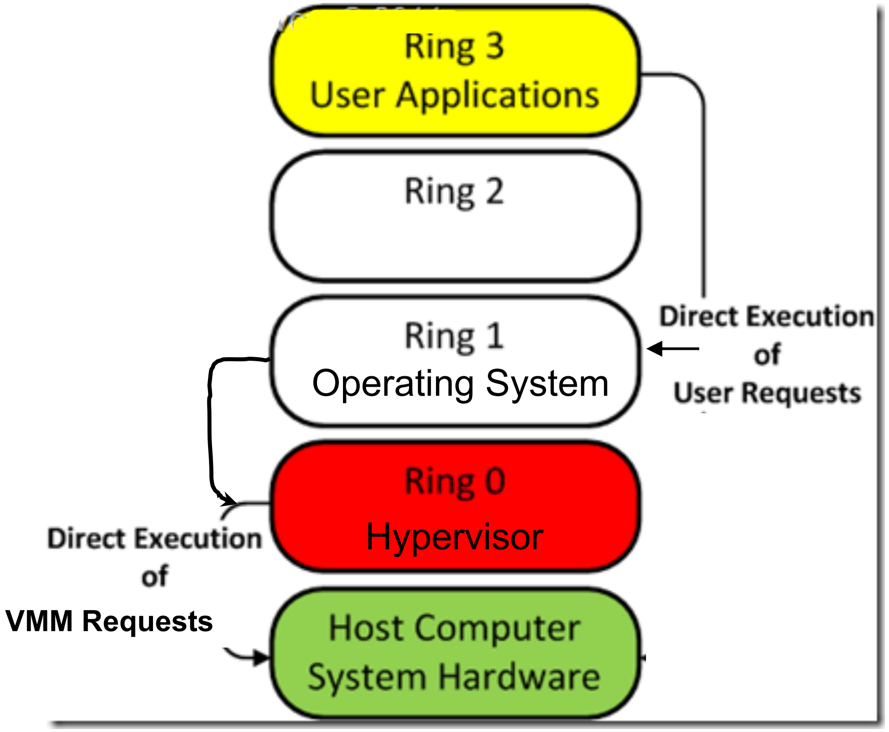


Virtualization types

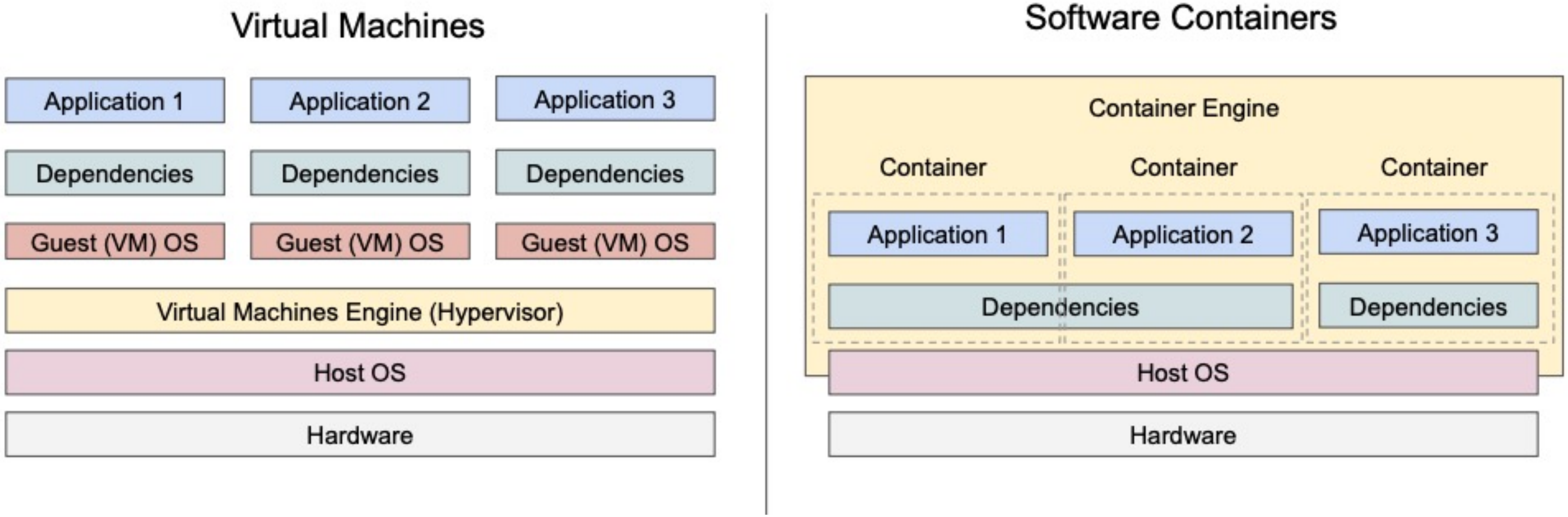
Full Virtualization: the hypervisor provides complete hardware abstraction creating simulated hardware devices. The guest OS don't know (or care) about the presence of a hypervisor and issue commands to what it thinks is actual hardware.

Paravirtualization: para means partial. The guest OS is aware that it is a guest, it recognizes the presence of a hypervisor, and it has drivers to issue some commands, mainly I/O operations, directly to the host OS, more efficiently than inside a virtual environment. The guest OS must be modified

Hardware assisted virtualization: is a type of full virtualization where the microprocessor architecture has special instructions to aid the virtualization of the hardware. These hardware extensions help the hypervisor tackle complex tasks at the processor level rather than through software emulation



Containers vs VMs



Container Advantages

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, can 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.

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

Container Advantages

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

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 health 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

Container Advantages

Easy Versioning A new container can be packaged for each new application version including all needed dependencies, modules and libraries at the "right" version

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

Containers are just processes

```
[root@gen10-02 ~]# docker run --rm -d nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
1f7ce2fa46ab: Pull complete
9b16c94bb686: Pull complete
9a59d19f9c5b: Pull complete
9ea27b074f71: Pull complete
c6edf33e2524: Pull complete
84b1ff10387b: Pull complete
517357831967: Pull complete
Digest: sha256:10d1f5b58f74683ad34eb29287e07dab1e90f10af243f151bb50aa
Status: Downloaded newer image for nginx:latest
6074c0f589361c9d7ed6a35c632a2b604f5cf4bb54bd7033a1996dd5c4341631
[root@gen10-02 ~]# ps -fC nginx
UID          PID  PPID  C STIME TTY          TIME CMD
root       91326 91310  1 18:28 ?        00:00:00 nginx: master process
101        91372 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91373 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91374 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91375 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91376 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91377 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91378 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91379 91326  0 18:28 ?        00:00:00 nginx: worker process
101        91380 91326  0 18:28 ?        00:00:00 nginx: worker process
```

```
root     91326 91310  0 18:28 ?        00:00:00      \_ nginx: master process nginx -g daemon off;
101      91372 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91373 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91374 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91375 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91376 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91377 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91378 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91379 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91380 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91381 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91382 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91383 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91384 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91385 91326  0 18:28 ?        00:00:00          \_ nginx: worker process
101      91386
```

Just a process

```
$ ps -ef –forest
```

```
[root@gen10-02 ~]# ls /proc/
1      13822  185    24797  27134  283    30595  360    3855   430    48598  5670   7052   8295   91413  buddyinfo
10     13868  186    248    27151  28411  306    361    3856   43006  486    5674   7073   82972  91414  bus
100    13870  187    24876  272    285    307    362    3857   43033  487    56775  7075   83     91415  cgroups
101    139    189    25     27208  28574  308    363    3859   431    488    56979  72     8313   91416  cmdline
103    14     19     250    27209  28575  309    365    386    432    48988  57     72213  83733  91417  consoles
104    140    190    2505   273    28580  310    366    3860   433    49     57020  72217  83951  91418  cpuinfo
105    141    191    251    274    28592  311    36619  3862   435    490    57810  72804  84     91419  crypto
106    142    192    252    27411  286    312    367    3864   436    49046  58     74     8432   91420  devices
108    144    194    253    27451  28619  313    368    3865   437    495    5827   74316  85     91421  diskstats
109    145    196    25306  27452  28620  315    36926  3866   438    496    5834   75     85093  91422  dma
11     146    1966   25425  27453  28625  316    37     3867   44     497    58755  76     86     91423  driver
110    14693  197    2549   27455  28679  317    370    3869   440    49727  5898   76136  86097  91424  execdomains
111    14698  198    255    27472  287    31772  371    387    441    498    59     76996  86580  91425  fb
11179  147    2      256    27474  28700  318    372    3870   4416   499    5909   77     86606  91426  filesystems
11180  14727  20     25625  27477  28726  31806  37229  3875   442    4998   59570  78     87261  91427  fs
113    149    200    257    27497  28727  31813  373    3879   443    5      598    7875   87719  91428  interrupts
114    14915  201    258    275    28730  32     374    388    445    50     5989   7876   88     91429  iomem
115    15     202    260    27512  288    320    375    3880   446    500    599    7877   89     91430  ioports
116    150    203    261    27524  28959  32014  376    3886   447    5005   60     78787  89071  91431  ipmi
118    15068  205    262    27543  28969  321    377    389    448    5007   600    79     89609  91432  irq
119    151    206    263    27559  28988  322    378    3894   449    5009   601    8      9      91433  kallsyms
12     152    207    264    27576  28991  323    37863  39     45     501    602    80     90     91434  kcore
120    15272  208    265    27580  29     32369  38     390    450    5010   60244  8013   90519  91435  keys
121    15274  209    2658   276    290    325    380    39052  45067  502    604    8023   91     91436  key-users
123    1536   210    266    27616  29022  326    3802   391    45087  503    605    8025   91291  91437  kmsg
124    154    21012  26622  27660  29023  327    3803   392    451    504    6076   80322  91310  91438  kpagecount
125    15488  211    26624  27682  29024  328    3805   393    452    5040   6078   8033   91326  91439  kpageflags
126    155    212    26626  27684  29034  33     3807   394    453    50440  61092  8051   91372  91440  loadavg
```

Interacting with the container as a process

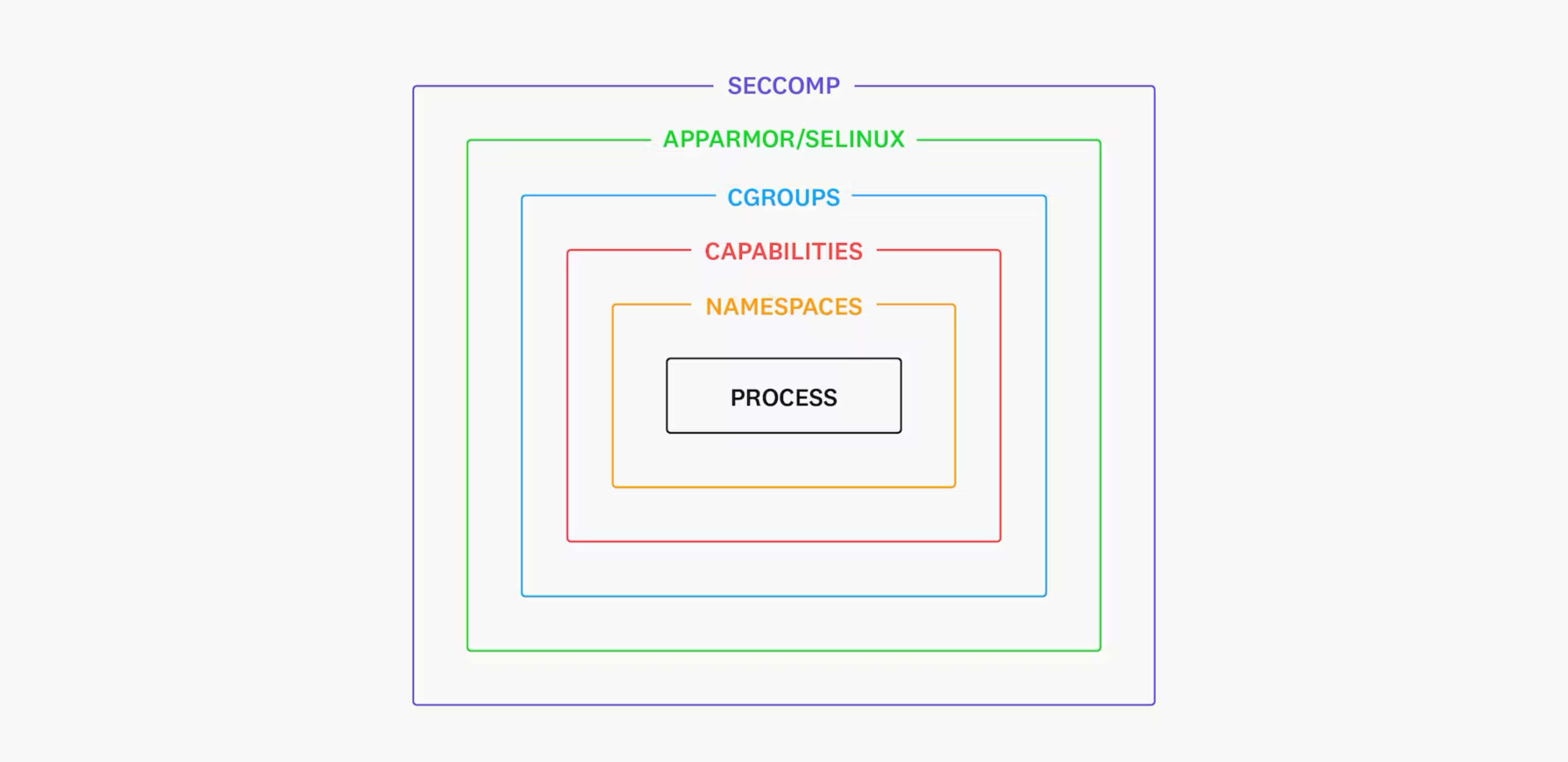
- The `/proc` filesystem in Linux is a virtual or pseudo filesystem. It doesn't contain real files—instead, it is populated with information about the running system.

This is the / of the container

```
[root@gen10-02 ~]# cd /proc/91326/root/
[root@gen10-02 root]# ls
bin   dev                   docker-entrypoint.sh  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  docker-entrypoint.d  etc                   lib   lib64  media   opt  root  sbin  sys  usr
[root@gen10-02 root]#
```

How does container works?

- How do we make sure that a process running in one container can't easily interfere with the operation of another container or the underlying host?



Linux namespaces

- Linux namespaces allow the operating system to provide a process with an isolated view of one or more system resources. Linux currently supports eight namespaces: Mount, PID, Network, Cgroup, IPC, Time, UTS, User

- NSs restrict a contained process's view of the rest of the host.

Namespaces

```
[root@gen10-02 ~]# lsns
        NS TYPE   NPROCS   PID USER COMMAND
4026531836 pid       921     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 22
4026531837 user     1018     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 22
4026531838 uts       921     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 22
4026531839 ipc       921     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 22
4026531840 mnt       917     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 22
4026531856 mnt         1   495 root kdevtmpfs
4026532052 net       921     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 22
4026533735 mnt         1 27018 ntp  /usr/sbin/ntpd -u ntp:ntp -g
4026533736 mnt         2 27058 root /usr/sbin/NetworkManager --no-daemon
4026533752 mnt        97 91326 root nginx: master process nginx -g daemon off
4026533753 uts        97 91326 root nginx: master process nginx -g daemon off
4026533754 ipc        97 91326 root nginx: master process nginx -g daemon off
4026533755 pid        97 91326 root nginx: master process nginx -g daemon off
4026533757 net        97 91326 root nginx: master process nginx -g daemon off
```

Role of NSs

- The mount (`mnt`) namespace provides a process with an isolated view of the filesystem. It can be useful for ensuring that processes don’t interfere with files that belong to other processes on the host. When using the mnt namespace, a new set of filesystem mounts is provided for the process in place of the ones it would receive by default.
- The `PID` namespace allows a process to have an isolated view of other processes running on the host. Containers use PID namespaces to ensure that they can only see and affect processes that are part of the contained application.
- network (`net`) namespaceis responsible for providing a process's network environment (interfaces, routing, etc.). It is very useful for ensuring that contained processes can bind the ports they need without interfering with each other, and for verifying that traffic can be directed to specific applications.
- Control groups (`cgroups`) are designed to help control a process's resource usage on a Linux system. In containerization, they’re used to reduce the risk of “noisy neighbors” (containers that use so many resources that they degrade the performance of other containers on the same host).

Linux Capabilities

- Capabilities split up the monolithic root privilege into 41 (at the time of publication) privileges that can be individually granted to processes or files.

```
[root@gen10-02 ~]# /usr/bin/pscap
ppid  pid   name        command           capabilities
27455 5396  root        sshd              full
5396  5398  root        bash              full
1     5518  root        screen            full
5518  5519  root        bash              full
5519  5622  root        perl              full
5622  5648  root        bash              full
5648  5651  root        beegfs-ctl/Main   full
5622  5670  root        bash              full
5670  5674  root        beegfs-ctl/Main   full
5622  5898  root        bash              full
5898  5909  root        beegfs-ctl/Main   full
27455 6076  root        sshd              full
6076  6078  root        bash              full
1     6192  root        screen            full
1     28726 root        slurmd            full
1     28727 root        agetty            full
1     28730 root        agetty            full
[root@gen10-02 ~]# /usr/bin/pscap | grep -i nginx
10438 10456 root        nginx             chown, dac_override, fowner, fsetid, kill, setgid, setuid, setpcap, net_bind_service
, net_raw, sys_chroot, mknod, audit_write, setfcap
```

You may be able to drop some or all of these capabilities to help harden your containers (`setcap`).

Cgroups basics

- Control groups (`cgroups`) are designed to help control a process's resource usage on a Linux system.

```
rorym in cuilean in kadmin@kubeadm2node () ~ took 26s
❯ lscgroup
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/sys-fs-fuse-connections.mount
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/sys-kernel-config.mount
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/sys-kernel-debug.mount
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/dev-mqueue.mount
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/user.slice
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/user.slice/user-1000.slice
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/user.slice/user-1000.slice/user@1000.service
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/user.slice/user-1000.slice/user@1000.service/user.slice
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/user.slice/user-1000.slice/user@1000.service/app.slice
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/user.slice/user-1000.slice/user@1000.service/app.slice/dbus.socket
cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/user.slice/user-1000.slice/user@1000.service/app.slice/dbus.service
```

Cgroups basics: limit CPU usage for a process

```
$ cgcreate -g cpu:/cpulimited
$ cgcreate -g cpu:/lesscpulimited
```

cpu controller has a property known as cpu.shares. It is used by the kernel to determine the share of CPU resources available to each process across the cgroups. The default value is 1024.

split the CPU resources using a 2:1 ratio

```
$ cgset -r cpu.shares=512 cpulimited
```

```
$ cgexec -g cpu:cpulimited /usr/local/bin/matho-primes 0 9999999999 >
/dev/null &
```

Cgroups basics: limit access to resources

- Docker offers various options for limiting the amount of CPU time the container can utilize, but the simplest is the --cpus flag,

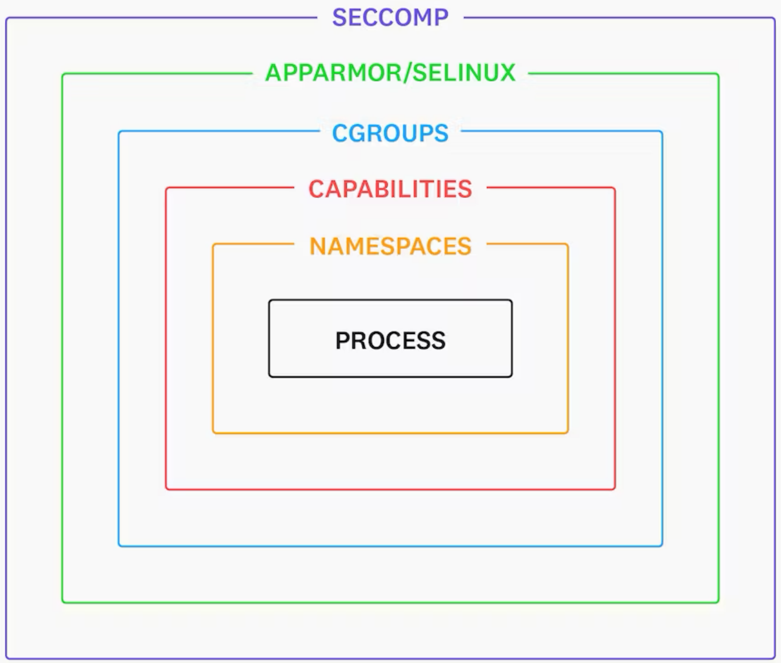
```
docker run --name stress --cpus 0.5 -it stressimage /bin/bash
```

- A common denial-of-service attack on Linux systems is known as a fork bomb:
  - cgroups can restrict the number of processes that can be spawned

```
docker run -it --pids-limit 10 ubuntu:22.04 /bin/bash
```

How does container works?

- Containers are just processes running on your host
- They share the same kernel (OS)
- They are isolated by NS+CG



Container Engines

1. Docker

1. Evolution from a monolithic project to supporting open-source ecosystems.
2. Docker Engine (open source for Linux) vs. Docker Desktop (freemium for Mac and Windows).
3. Root access required (containerd)

2. Podman

1. Daemonless container engine for managing OCI Containers.
2. Offers root and rootless mode for security and usability.
3. User ID (UID) management issues in rootless mode.

3. Containerd and CRI-O

1. Containerd serves as a runtime, not intended for direct usage but can be accessed via Containerd CLI (ctr).
2. CRI-O behaves similarly, not meant for direct command-line usage; crictl for debugging.

Container Engines

Singularity

Functions more like a virtual environment than a container engine.

Limited isolation between containers and host, affecting security and behavior.

Shares directories, environment variables, and lacks proper user mapping.

LXD and RKD

LXD manages both containers and virtual machines for full Linux systems.

RKD (Rocket) was a command-line utility, now an ended project for running containers using kernel-level calls.

Container Orchestrators

1. Docker Compose

1. Creates multi-service applications on a single node.
2. Uses a docker-compose.yml file to assemble containers with a dedicated network.
3. Supports only Docker APIs; Podman can work with it by emulating Docker.

2. Docker Swarm

1. Manages multi-node deployments in a cluster of Docker engines called a "swarm."
2. Similar to Docker Compose but suitable for small teams or simple deployments.
3. Supports only Docker APIs.

3. Kubernetes

1. Full-featured container orchestration solution for various settings and network topologies.
2. Supports multiple container runtimes, dropped support for Dockershim in favor of Containerd.
3. Introduces the concept of a "pod" to the container ecosystem.
4. Challenging to master; can be accessed via CLI and REST APIs.

Container Orchestrators

1. **AWS ECS (Elastic Compute Service)**
    1. Amazon's internal implementation similar to Kubernetes.
    2. Requires Docker Engine for managing Amazon VMs or offers pre-built VM images.
    3. Uses Docker Engine, not a container runtime.
2. **AWS Fargate**
    1. A serverless execution of containers on AWS infrastructure.
    2. Eliminates concerns about underlying OS/hardware.
    3. Shifted from Docker Engine to Containerd with platform version 1.4 in April 2020.

How to run a container

Get or build a container image (think about it as a file)

Run the image: this is your container

```
docker run my_container
```

..where “docker” can be replaced with your container engine of choice, e.g Podman.

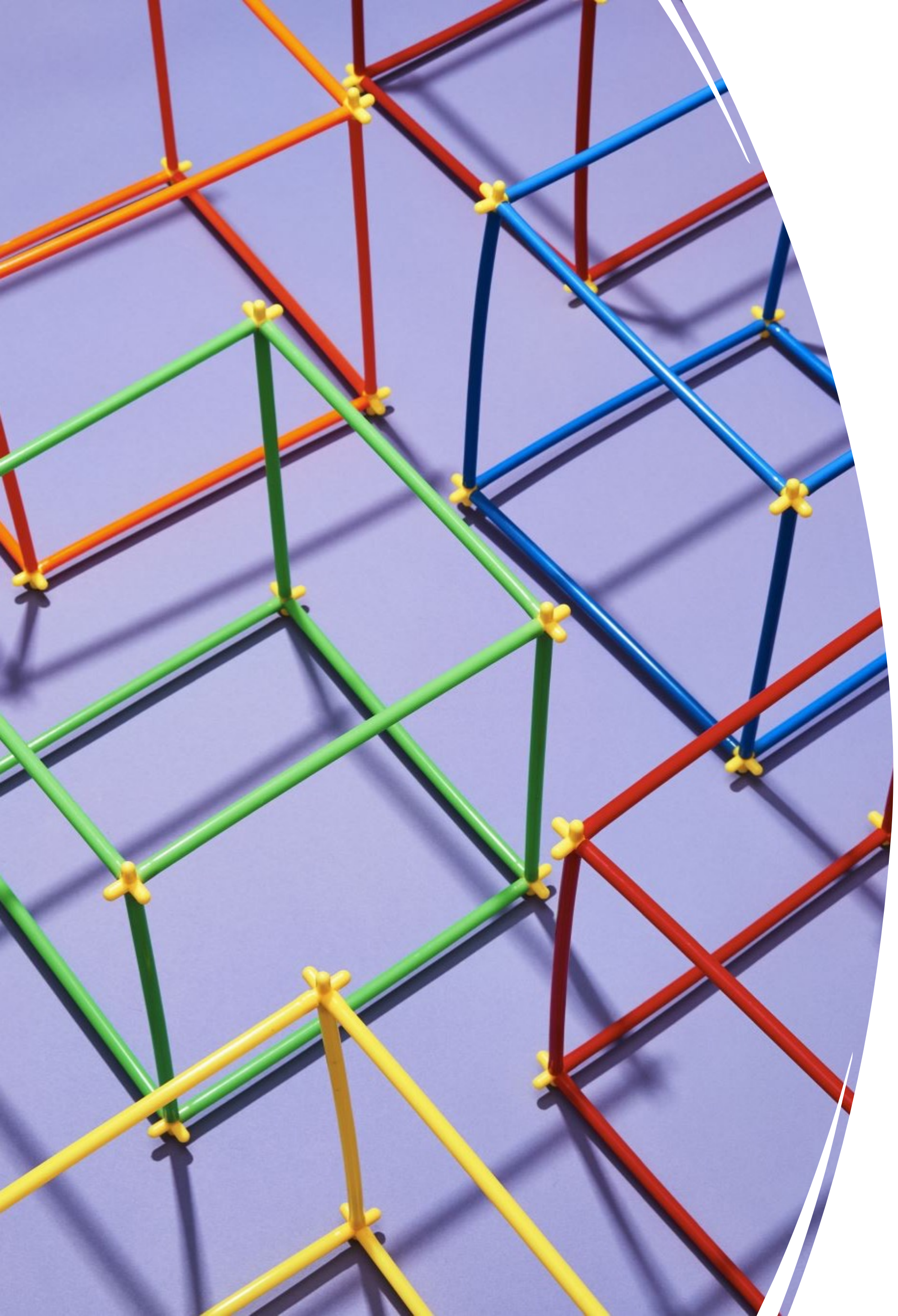
Note: many engines, if cannot find the image locally, will automatically look online.

Example:

```
$ docker run hello-world

Unable to find image 'hello-world:latest' locally latest:
Pulling from library/hello-world

2db29710123e: Pull complete

Digest:
sha256:6d60b42fdd5a0aa8a718b5f2eab139868bb4fa9a03c9fe1a59ed494
6317c4318

Status: Downloaded newer image for hello-world:latest


Hello from Docker!

This message shows that your installation appears to be
working correctly.
```

Example:

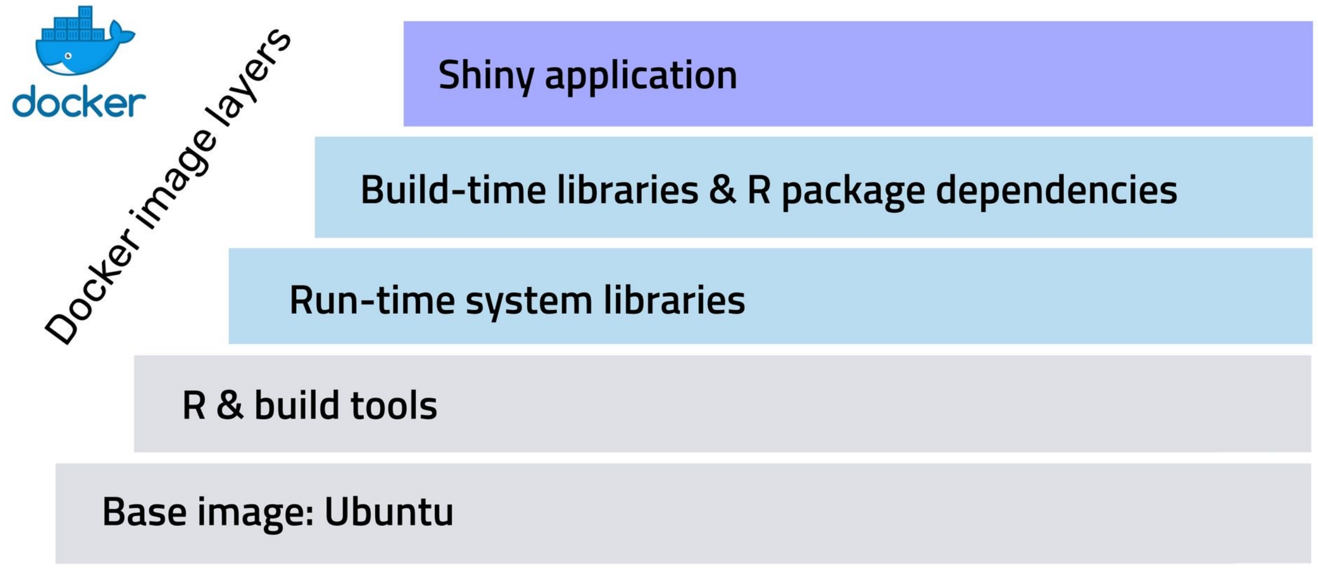
- `$ docker run -it python:3.8`
- `$ docker run --entrypoint /bin/bash -it python:3.8`



Main Concept: Images

- A container image is a static file with executable code that can create a container;
- A container image is immutable—meaning it cannot be changed, and can be deployed consistently in any environment;
- A container is the running version of an Image;
- A container is volatile: once closed it is destroyed.

Main Concept: Images



- When you define a Docker image, you can use one or more layers, each of which includes system libraries, dependencies and files needed for the container environment.
- Image layers can be reused for different projects.

Images

```
[root@gen10-02 ~]# docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
python              latest              58a8f3dcd68a        4 days ago          1.02GB
ubuntu              latest              b6548eacb063        11 days ago         77.8MB
nginx               latest              a6bd71f48f68        3 weeks ago         187MB
python              3.8                 8a61cde23424        7 weeks ago         997MB
bp_sim              latest              0329a2e2d677        2 years ago         985MB
<none>              <none>              48762b1760fe        2 years ago         1.02GB
<none>              <none>              210634d66eec        2 years ago         1.07GB
[root@gen10-02 ~]#
```

Images

```
[root@gen10-02 ~]# docker inspect 58a8f3dcd68a
[
    {
        "Id": "sha256:58a8f3dcd68a25102665617db6b9cc605dac7e5b84a874c456692513d12c990f",
        "RepoTags": [
            "python:latest"
        ],
        "RepoDigests": [
            "python@sha256:6d7fa2d5653e1d0eb464a672ded01f973e49e4a7ded59703c7bdcf6b92eac736"
        ],
        "Parent": "",
        "Comment": "buildkit.dockerfile.v0",
        "Created": "2023-12-08T04:49:21Z",
        "Container": "",
        "ContainerConfig": {…
```

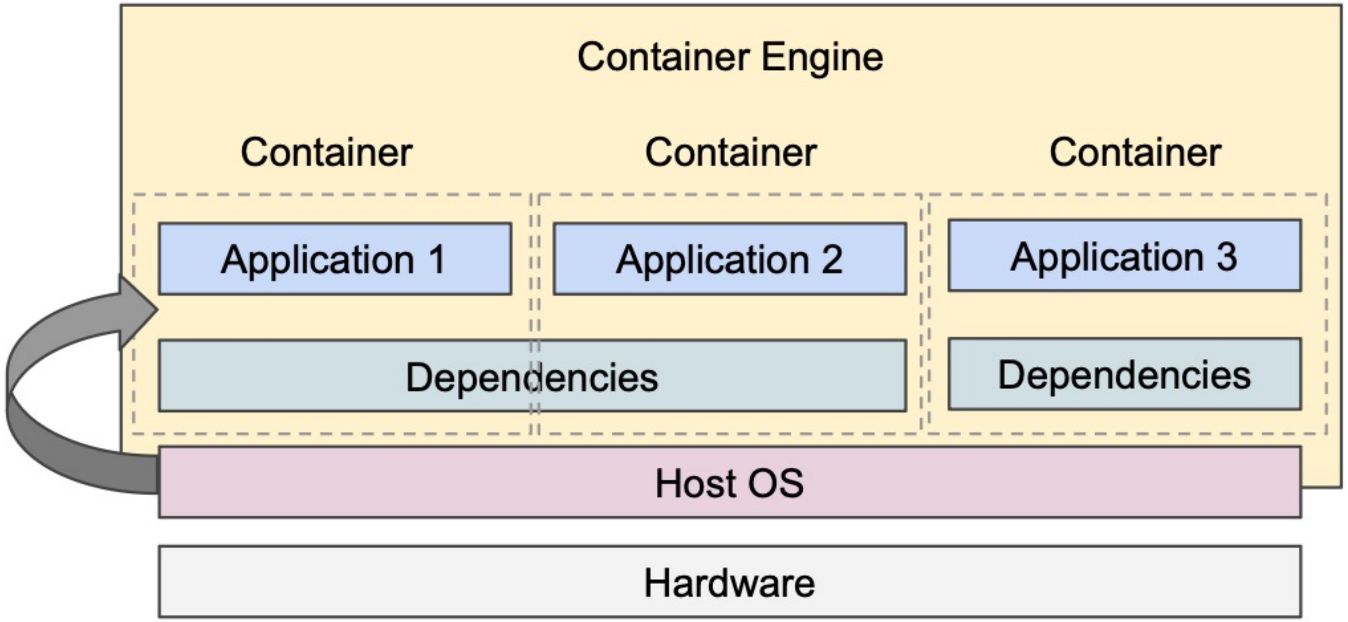
Images


```
"RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:7cea17427f83f6c4706c74f94fb6d7925b06ea9a0701234f1a9d43f6af11432a",
                "sha256:7c32e0608151e1683f9e1ee78eb507fe9fe73fc10584fc5f09b6b0475b95871b",
                "sha256:30f5cd833236dc35f9ab67c205f913fc238902ee71f28a47fdb7d1ecaa9f0776",
                "sha256:80bd043d4663600a3f8fa3d36604acf9885cef3f67b8a878d510c07289761972",
                "sha256:2c8a14dec1261f607a90c797e90fb81be2b6eb945549f7af8ea518e9b47a53ec",
                "sha256:e62b338f9de766465c65156a2eb17ff5a861d30430f11bfe251271f6777d4695",
                "sha256:77b3c1d92acfc6e716a2de090ff4477fc2687eb67df752a82777c0e2acc31f6f",
                "sha256:8a4ac491bc4903a9c7e758c4c705110399f2608b508a92d2dad023a76b583856"
            ]
        "Architecture": "amd64",
                "Os": "linux",
          "Size": 1018287322,
                "VirtualSize": 1018287322,
                "GraphDriver": {
```

Main Concepts: Volumes

How to share files with a container? → ***volumes***

We are creating a bridge



Example

- Make your home folder visible from within a container

```
$ docker run -it -v $HOME:/data python:3.8
```

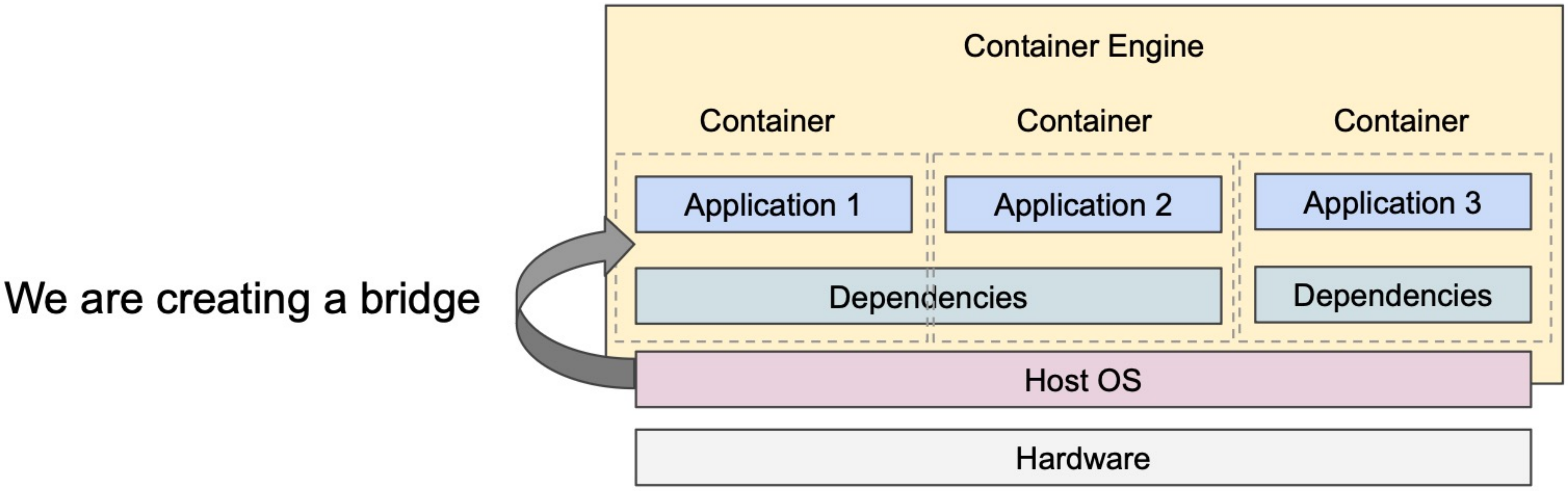
```
$ docker run -v $HOME:/data  --entrypoint /bin/bash -it python:3.8
```

Example

```
$ docker run -it -v $HOME:/data python:3.8
Python 3.8.12 (default, Dec 21 2021, 10:45:09)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>> import os
>>> os.listdir('/data’)
['Applications', 'Desktop','Documents',
'Downloads', 'Dropbox', 'iCloud', 'Library',
'Movies', 'Music', 'Pictures', 'Public']
```

Main Concepts

- How to access servers* in a containers? → ***port mapping***



Examples

```
$ docker run -p 9001:8888 jupyter/tensorflow-notebook:tensorflow-2.4.1
```

```
$ docker run -p 9002:8888 jupyter/tensorflow-notebook:tensorflow-2.4.3
```

Main Concepts: Performace aspects

```
$ python3 -m unittest discover
..................................................................................
........ ----------------------------------------------------------------------
Ran 90 tests in 41.405s
```

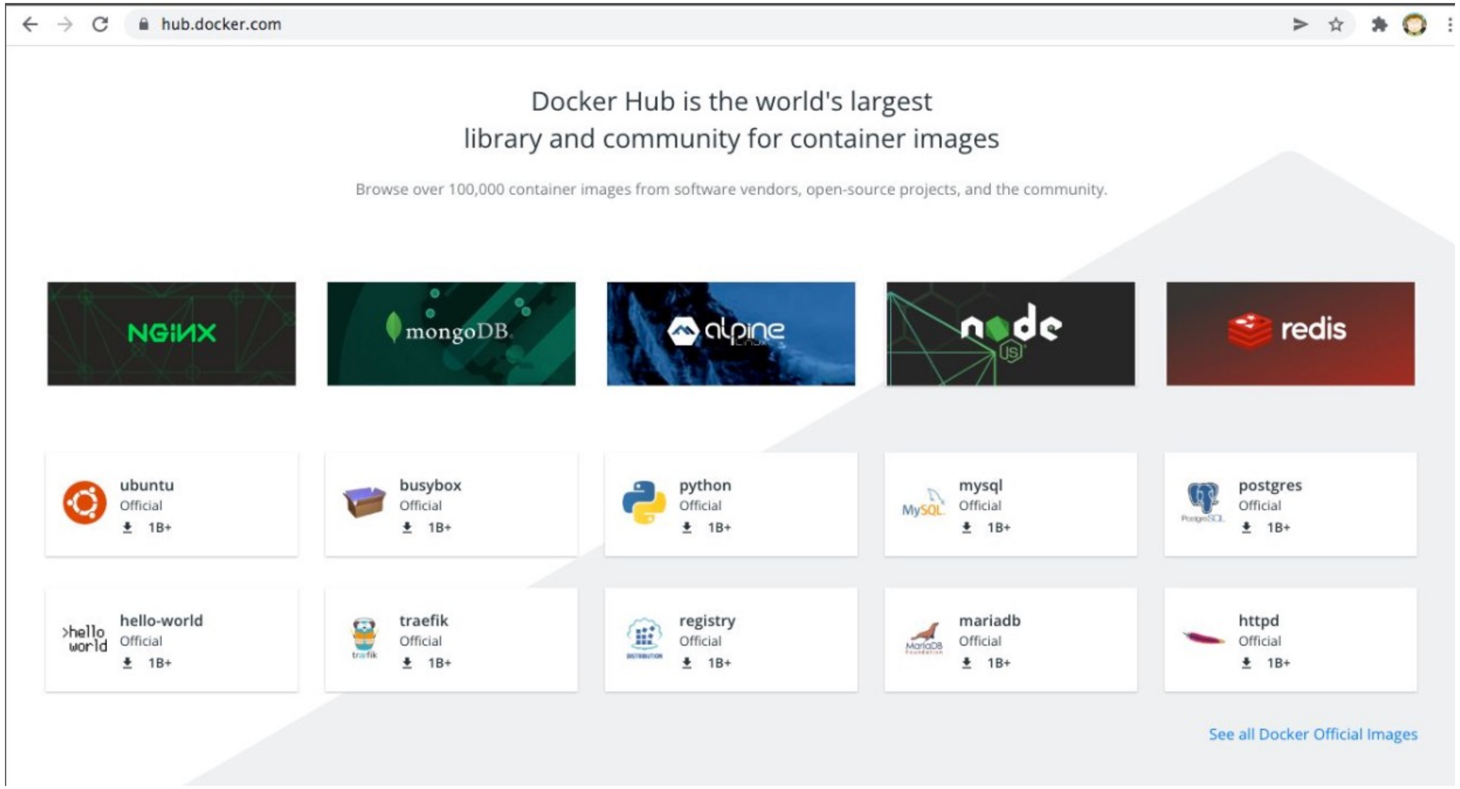
Host

```
$ python3 -m unittest discover
..................................................................................
........ ----------------------------------------------------------------------
Ran 90 tests in 34.108s
```

Container

Main Concepts

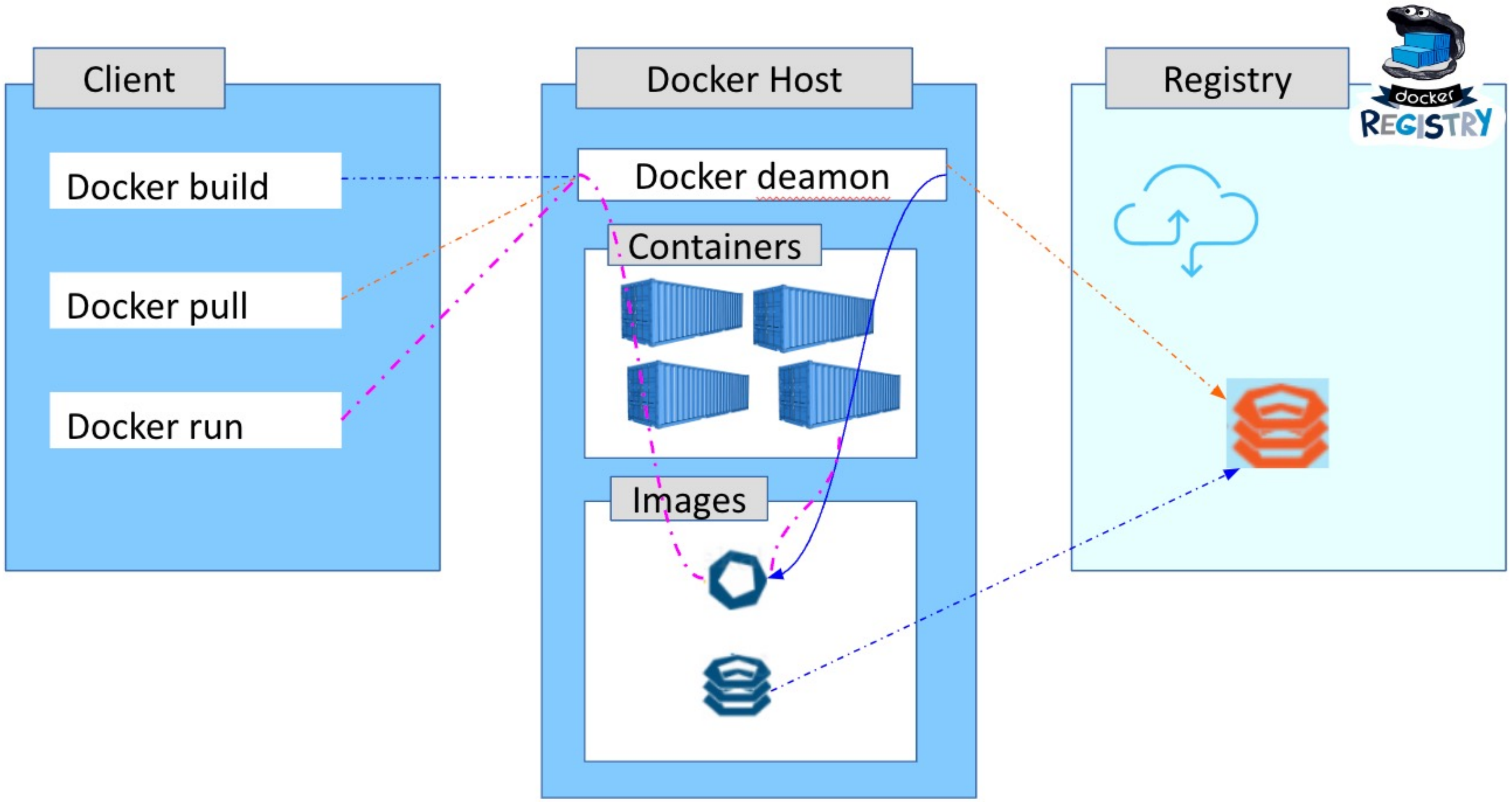
How to share containers?
→ ***registries*** (the GitHub for software containers)



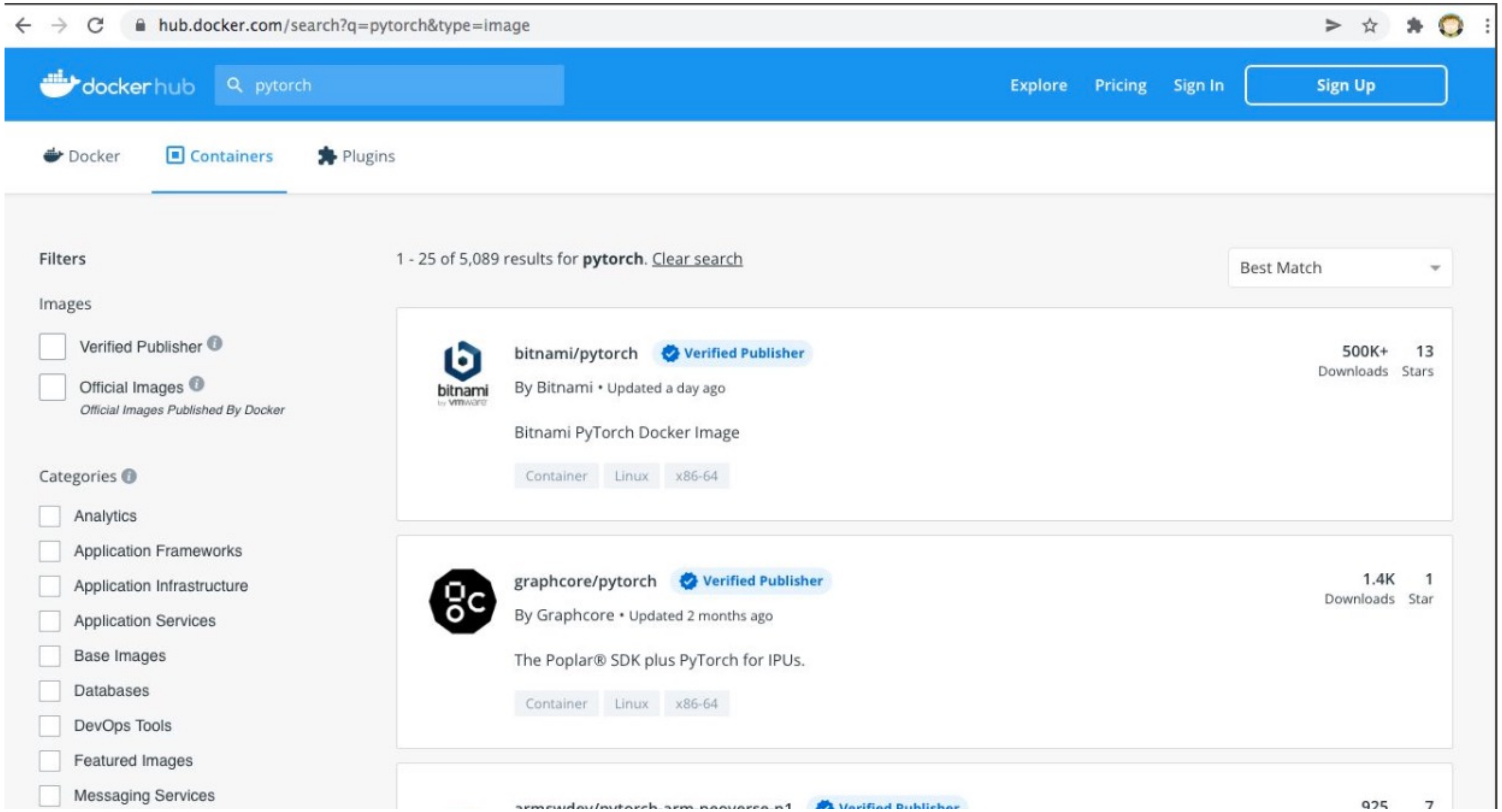
The Image Registry

- Docker Registry is where the Docker Images are stored. The Registry can be either a user's local repository or a public repository like a Docker Hub allowing multiple users to collaborate in building an application. Even with multiple teams within the same organization can exchange or share containers by uploading them to the Docker Hub.
- Docker Hub is Docker's very own cloud repository similar to GitHub.

The docker architecture: registry



DockerHub Registry



Docker

- Modern containerization solution, open source + freemium
- Extremely popular, the “de facto” containerization standard
- Incremental File System
- Plenty of software on Docker Hub
- Native on Linux
- Almost native on Macs post-2011 and Windows 10 (through a light VM)

→ Issues with new Apple M1 (ARM) chips!

Docker

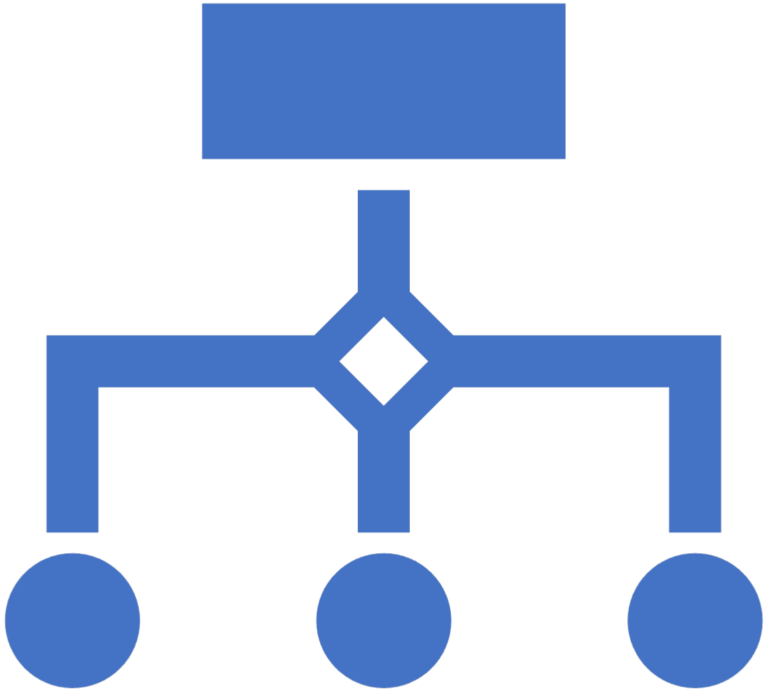
- Relies on a system daemon to manage containers
- Running containers are seen as (micro)services
- Containers have an IP address by default
- Extensive support for networking between containers
- Requires a privileged user (do not expect to do a “docker run” on clusters)
- Loads of orchestrators (docker-compose, kubernetes..)

Docker commands

- ***docker build:*** Build a container
- ***docker pull:*** Pull a container (from a registry)
- ***docker run:*** Run a container (and execute the default command, or a custom one)
- ***docker ps:*** List running containers
- ***docker exec:*** Run a command in a running container
- ***docker stop:*** Stop a running container
- ***docker rm:*** Remove a container
- ***docker image rm:*** Remove an image
- ***docker image ls:*** list local images

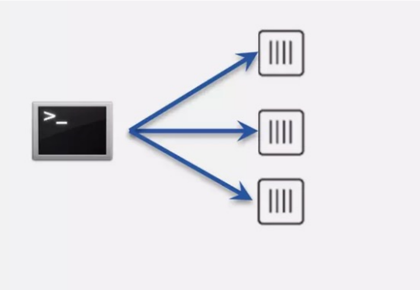
Orchestratio: docker-compose

- **Docker Compose** is a tool that allows you to run multi-container application environments based on definitions set in a YAML file.
- It uses service definitions to build fully customizable environments with multiple containers that can share **networks** and data **volumes**.

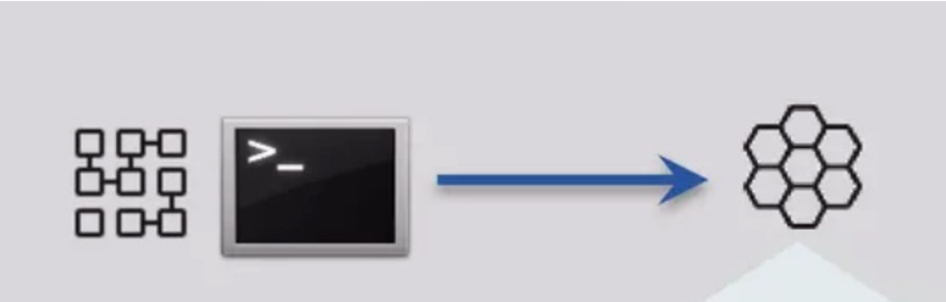


Running complex apps

- Without compose
  - Build and run one container at a time
  - Manually connect containers together
  - Must be careful with dependencies and start up order



- With compose
  - Define multi container app in compose yml file
  - Single command to deploy entire app
  - Handles container dependencies
  - Works with Docker Swarm, Networking, Volumes, Universal Control Plan



What is docker-compose?

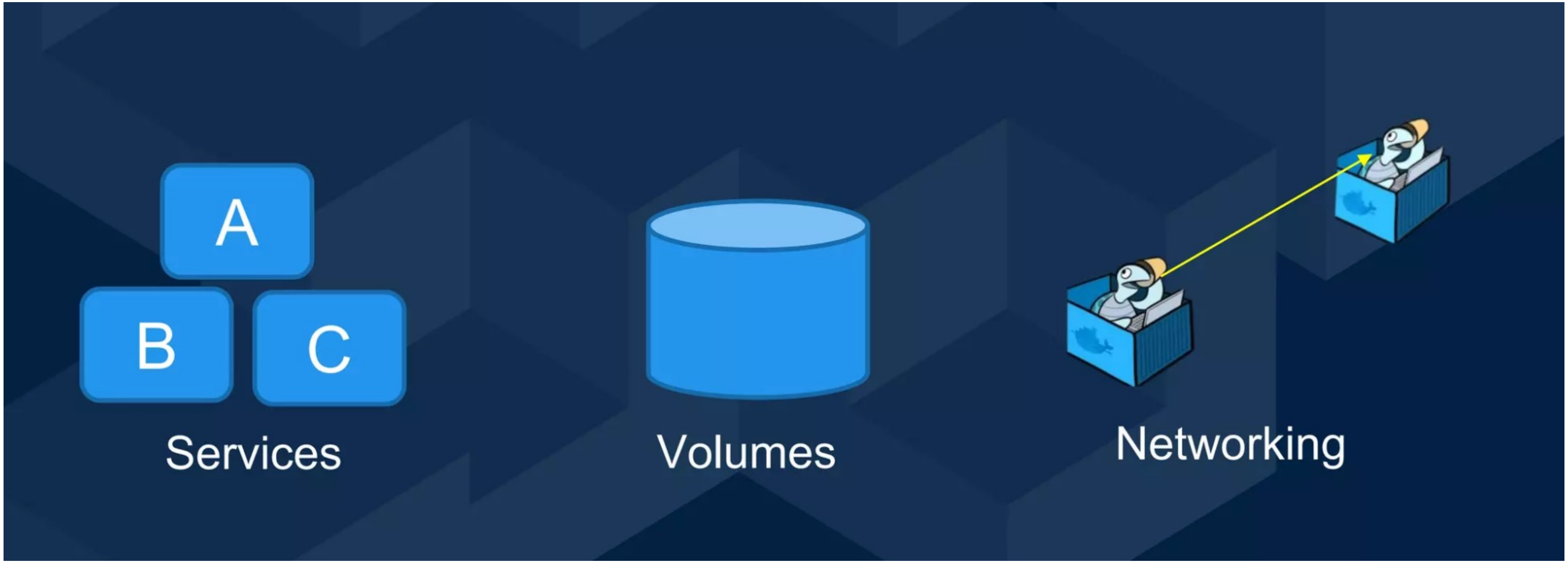
A tool for defining and running multi- container Docker applications

With Compose, you use a YAML file to configure your application's services

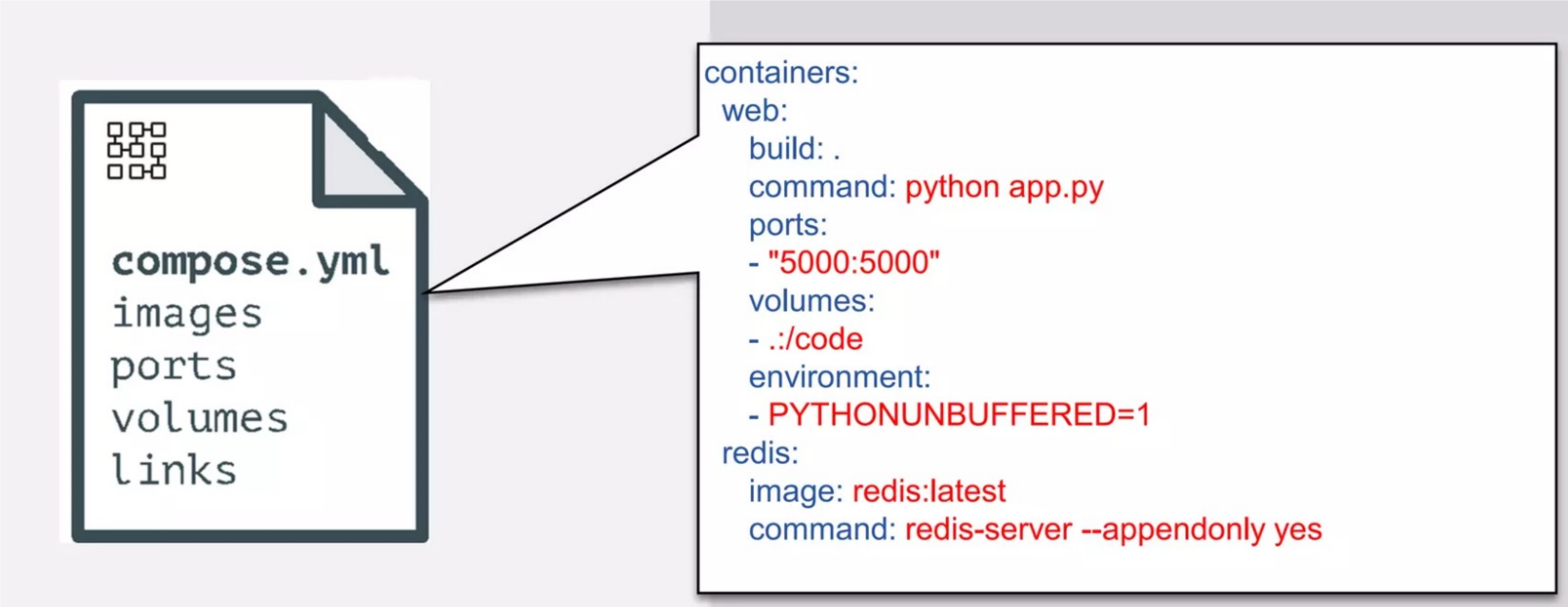
Compose works in all environments: production, staging, development, testing, as well as CI workflows.

With a single command, you create and start all the services from your configuration

Bulding blocs of compose



Container orchestration with compose




```
$ docker compose up
```

Questions