Lab 4: Containerization

Cloudbase Solutions AC Labs 4

Cloudbase Solutions SRL

Liga AC Cloudbase Labs 4: Containerization

Cloudbase Solutions

June 7, 2023

Abstract

This document, which is based on the contents of the Containerization Liga AC LABS 2023 session held by Cloudbase Solutions, is meant to act as a "begginer's guide" to the history and inner workings of Containers on Linux operating systems.

The main takeaway for the Reader should be how Linux Containers are simply the logical composition of process isolation techniques into a single solution for releasing and running your application code in a predictable manner.

We will walk you through how various aspects of processes (such as the files they have access to and the system processes they can interact with) are restricted or downright isolated from the Container's processes, as well as seeing how a Container Image is created and run.

After some initial theoretical considerations, this document will focus on applying all the theory through numerous Bash command listings which we encourage the Reader to follow along with on their own Linux system.

An additional section containing tasks geared towards the Reader applying the knowledge on their own will be included at the end as well.

Contents

Listings:

Chapter 1

Why Containers?

As the Cloud Computing industry progressed, the practice of offering Virtual Machines to customers for them to install and run their workloads, which came to be known as **I**nfrastructure **a**s **a** **S**ervice (or **IaaS**), became the standard business model industry-wide.

It quickly became apparent, however, that most Ops divisions were only ever running a *single service per Virtual Machine*, with said service often comprising of just one single actual system process, such as a non-child-process-forking HTTP server for example. To further add insult to injury, the Guest Operating System which customers ran within their Virtual Machines was more often than not *the same as the Host OS*. (i.e. similar Linux kernel versions or Windows Server builds as the Host)

The question then naturally arose: would it be possible to simply run the customers' services as "contained processes" directly on the Host Operating System while offering the same security and power of a dedicated Virtual Machine?

Figure 1.1: Containerization vs. Virtualization (Meme).

Well the short answer is no, Hardware-Assisted virtualization through Type 1 Hypervisors provides hardware-level security assurances and will thus always be more secure than a Containerized Process, but we can get more than close enough to make both customers and managers alike happy.

1.1 Comparing VMs and Containers.

In this section, we'll quickly run through a high-level overview of the differences between *virtualized apps* (i.e. processes running in VMs) and *Containerized apps*. (i.e. processes running in an isolated container)

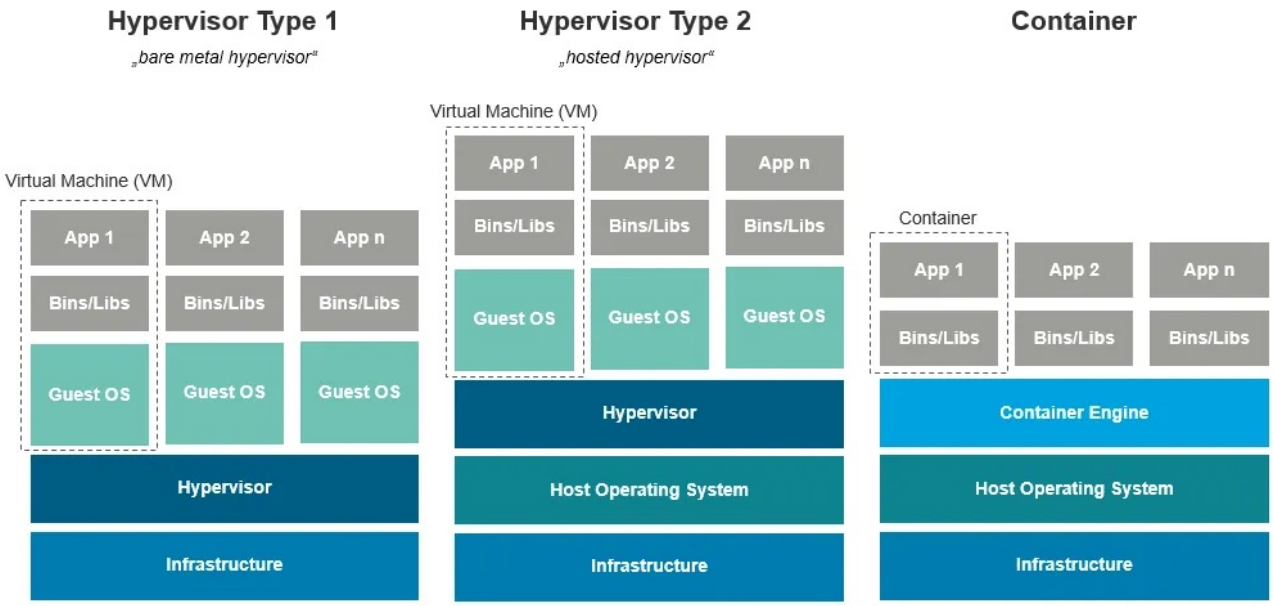

Figure 1.2: Containerization vs. Virtualization.

Looking at the bigger picture, it is notable that **no one solution can be named the best**, as each presents both upsides and downsides when compared with any other. It depends on what the Ops team needs at the time for the given usecase or customer requirements:

- Hardware-Assisted Virtualization (Type 1):
  - best when security is paramount and added special hardware costs are acceptable.
  - most secure by far, as it leverages special CPU features which basically **guarantee security at the hardware level**. This makes them the only option for high-security environments like the **Fin**ancial **Tech**nology (FinTech) industry or government-related projects.
  - matching Host and Guest OSes (e.g. only running Windows VM on Windows Hyper-V Hosts) is **not** a requirement, but is usually recommended for improved compatibility or even special features. (e.g. app-consistent backups, where apps like Microsoft MSSQL in VMs can be notified by the Host to flush boffers to ensure a disk snapshot is "consistent")
- Hardware Emulation (Type 2):
  - slowest by far since all the Virtual Machine's hardware needs to be emulated in software.
  - is the only viable option for cross-CPU-architecture solutions. (e.g. emulating an ARM-based Android phone in Android Studio on your x86 laptop, or cross-building a container image for the PowerPC architecture on an ARM server)
- Containers:
  - fastest for usecases where the Guest OS and Host OS can be the same and the containerized Guest Application is compatible with the Host's kernel version.

– least secure by far, as it relies on Host-kernel-level isolation features, so it should **not** be used in any security-critical domains such as Government-related projects.

We could sum it all up in the following table:

Technology	Performance	Security	Guest CPU Lim.	Guest OS Lim.	Extra Limitations
Type 1	fast	maximum	match host	none	special hardware
Type 2	slowest	good	none	none	none
Containers	fastest	low	match host	match host	match host kernel ver.

Chapter 2

What needs Containerizing?

In this chapter we'll look at what capabilities of a standard Linux process need to be isolated or even forbidden completely in order to be able to run said process as a Container.

The Linux kernel offers a large number of services for its processes:

- **Filesystem access**: all processes can open and close files freely (assuming the appropriate permissions are satisfied), so the first logical step to running containers in an isolated fashion is completely isolating their filesystem. This is achieved through the `pivot_root` functionality and the so-called **mount namespaces** features within the Linux kernel.
- **Inter-process communication**: another requirement of running containers would be to have the containerized processes completely isolated from the actual Host system processes. Containers should *not* be able to see or interact with any of the main system processes, and process IDs (PIDs) should be abstracted per-container, so any container can have any process IDs internally. This requires the addition of a couple of features in the Linux kernel called **process namespaces** (or **PID namespaces**) to isolate process IDs, and **IPC namespaces** to isolate Inter-Process Communication.
- **Networking**: in order to run remotely-accessible services within containers, we need a way to isolate networking features such as IP addressing, routing table information, and even firewall rules for each individual container. This is achieved through **network namespaces**.
- **CPU/RAM resources**: while it would be nice for containerized processes to have access to all Host system resources in the same way that native processes can, having a container crash the Host because it asked the Host to allocate infinite RAM would be a security risk. For this, the Linux kernel offers a feature called **cgroups**, which allows the administrator to specify maximum CPU/RAM resources allowed for a containerized process and any child processes it may spawn within its process namespace.
- **Auxiliary system features**: things like the container's system clock (handled by **time namespaces**) and the hostname (handled by so-called **Unix Time Sharing (UTS) namespaces**) also need considering, and have their own dedicated solutions within the Linux kernel.

Let's look at each of these aspects individually in the following sections before we try to compose them all together.

2.1 Filesystem sandboxing with `chroot`.

In order to ensure that a containerized process cannot mess with the files of the Host Operating System, we need to set up a kind of "*filesystem jail*" (aka a "*sandbox*") where the process can freely create and modify a set of files which are separated (or *sandboxed*) from the Host's files. While we could set up this sandboxing for each individual file, it would be more handy if we could basically **mount** a whole directory structure to the root (`cd /`) of the filesystem of the new container.

```
# First off, let's do a quick recap of filesystem mounting on Linux.

# You can check what things your Host OS has mounted with "mount",
# or you can simply "cat /etc/mtab", as it produces the same results.
$ mount
# NOTE: the /sys and /proc pseudo-filesystems:
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
# NOTE: udev is the "device mapper" which populates /dev:
udev on /dev type devtmpfs (rw,nosuid,relatime,size=1434716k,nr_inodes=358679,mode=755)
# NOTE: this is your boot partition where GRUB the bootloader lives:
/dev/sda2 on /boot type ext4 (rw,relatime)
# NOTE: your root filesystem (may be a /dev/mapper/... path if using LVM)
/dev/mapper/ubuntu--vg-ubuntu--lv on / type ext4 (rw,relatime)
...

# You can even mount one specific directory into another through a "bind mount":
$ mkdir /tmp/bind_source_dir
$ mkdir /tmp/bind_target_dir
$ echo "I believe I have a unique path - said the file." > /tmp/bind_source_dir/file
$ sudo mount --bind /tmp/bind_source_dir /tmp/bind_target_dir

# It looks like it's the same file:
$ cat /tmp/bind_target_dir/file
I believe I have a unique path - said the file.

# "stat"-ing the file should tell use the exact device/inode details:
$ stat /tmp/bind_target_dir/file
  File: /tmp/bind_target_dir/file
Device: fd00h/64768d    Inode: 786456      Links: 1
$ stat /tmp/bind_source_dir/file
  File: /home/nashu/bind_source_dir/file
# NOTE: exact same device and inode number, so is exact same file.
Device: fd00h/64768d    Inode: 786456      Links: 1

# Let's try to setup a "filesystem sandbox" we can try to run a process in later.
# For this, we'll basically just copy over relevant directories from the Host OS:
$ sudo mkdir /tmp/sandbox
$ for dir in /bin /etc /lib* /sbin /usr /var; do sudo cp -r $dir /tmp/sandbox${dir}; done

# We can also create some empty directories we will need later:
$ for dir in /boot /proc /dev /mnt /root; do sudo mkdir -p /tmp/sandbox${dir}; done

# The end result should be a "filesystem-level" copy of most files on the host.
# Let's make a couple of files to mark each one so we don't get confused later:
$ sudo bash -c 'echo "This file is in the REAL root." > /root.txt'
$ sudo bash -c 'echo "This file is in the SANDBOX root." > /tmp/sandbox/root.txt'

# In the following examples, if you ever find yourself unsure whether
# you're in the container sandbox or not, you can simply:
$ cat /root.txt
This file is in the REAL root.
```

Listing 1: Mountpoints refresher and sanbox setup.

```
# Now that we have set up a nice little sandbox, we need to launch a process in it.
# To switch (or "pivot") to the new root, we can use the "chroot" command:
$ sudo chroot /tmp/sandbox /bin/bash

# NOTE: this document will use "[C]" to denote commands you should run INSIDE the "chroot"!

# We will now be in a "bash" shell whose "/" directory is actually "/tmp/sandbox".
# If we ls /, it will look like our new root, complete with the root.txt file:
[C] $ ls /
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  mnt  proc  ...
[C] $ cat /root.txt
This file is in the SANDBOX root.

# Listing all mounts with "mount" will claim there is no /etc/mtab:
[C] $ mount
mount: failed to read mtab: No such file or directory

# "ps" also has trouble listing processes, but it provides a helpful suggestion:
[C] $ ps
Error, do this: mount -t proc proc /proc
[C] $ mount -t proc proc /proc

# Running "mount" again will show us our new procfs mount:
[C] $ mount
proc on /proc type proc (rw,relatime)
# Note that because we mounted the /proc on the host, we can see all host processes:
[C] $ ps aux
<all the processes on your host>

# You may be wondering what's stopping us from simply mounting "/dev" in the chroot?
[C] $ mount -t devtmpfs udev /dev

# Wait a second, does that mean we can directly mount the actual Host root device?
[C] $ mkdir /realroot
# NOTE: please check what your real root device is by running "df" on your host.
[C] $ mount /dev/mapper/ubuntu--vg-ubuntu--lv /realroot
[C] $ cat /realroot/root.txt
This file is in the REAL root.

# Did we seriously just manage to beat our filesystem sandbox so easily?
# No, you see, chroot only patches the root of the filesystem (aka "/") for
# the "chroot" process WITHOUT applying any limitations to the user running it:
[C] $ whoami
root

# We can even note that our Bash shell is owned by the real "root" user on our system,
# so it can do all the things the real root can do. (such as mount devices from /dev)
[C] $ ps aux | grep $$
root        1603  0.0  0.3   8024  4052 ?        S    12:27   0:00 /bin/bash -i

# If only there were a mechanism through which we could...I dunno...
# map usernames in sets of named spaces or something like that maybe?
```

Listing 2: `chroot` intro an limitations.

2.2 User namespaces.

```
# While we do want our container to "feel like root", we absolutely do NOT want
# it mounting actual system devices, so we need to "namespace" the users in the
# containerized process so it thinks it's running as "root", but actually isn't.
# NOTE: please run "exit" in the "chroot" and run the following on the Host!

# Let's take note of the user namespace on the Host system:
$ sudo lsns -t user
        NS TYPE  NPROCS PID USER COMMAND
4026531837 user     116   1 root /sbin/init
# NOTE: the above is the "root user namespace", in which the Linux kernel
# started the "/sbin/init" init system. (aka PID 1, aka Systemd)
# This user namespace is used by all "normal process" on our system.

# In order to isolate our "chroot" process into its own User Namespace, we will
# use the "unshare" command, which will make the "chroot" NOT share the main namespace.
# "unshare" will automatically create a new user namespace for us with "-U".
$ sudo unshare -U chroot /tmp/sandbox /bin/bash
chroot: cannot change root directory to '/tmp/sandbox': Operation not permitted

# That's weird, what permission is it talking about? We're running it with "sudo"?!
# Well, because we had "unshare" execute the "chroot" command in a new user namespace,
# and there are no actual users defined within it, any process we try to run
# (such as "whoami" or the "chroot" itself)  will actually be running as "nobody":
$ sudo unshare -U whoami
nobody

# Even worse, all the files in /tmp/sandbox still belong to the real "root",
# which does not exist in the new User Namespace, and also shows up as "nobody":
$ ls -l /tmp/sandbox
lrwxrwxrwx   1 root root    7 Jun  2 12:24 bin -> usr/bin
drwxr-xr-x  21 root root 4080 Jun  2 12:30 dev
...
$ sudo unshare -U ls -l /tmp/sandbox
lrwxrwxrwx   1 nobody nogroup    7 Jun  2 12:24 bin -> usr/bin
drwxr-xr-x  21 nobody nogroup 4080 Jun  2 12:30 dev
...

# This means that our "chroot" process, running as "nobody", is not allowed to
# "cd /tmp/sandbox", since "nobody" has no permissions on the system.
# In order to have "unshare" automatically map the root user, we can use "-r":
$ sudo unshare -U -r chroot /tmp/sandbox /bin/bash
[C] $ lsns -t user
        NS TYPE  NPROCS   PID USER COMMAND
# NOTE: different user namespace ID than the host!
4026532162 user       2  1876 root /bin/bash

# We are now also properly prevented from doing root stuff in the sandbox:
[C] $ mount -t devtmpfs udev /dev
mount: /dev: permission denied.
```

Listing 3: User namespaces showcase.

2.3 Mount Namespaces.

While a solution for user-level isolation like User Namespaces is great for preventing users from mounting actual devices, it would still be nice to offer Containers the ability to set up mounts within them. (e.g. creating bind mounts in its own little mount sandbox if it wants to) For this, we'd need another Linux kernel feature to isolate the mounts each process can see called **mount namespaces**.

```
# If you've been following along with the previous sections, you may note
# /dev, /proc, and /realroot are still mounted within /tmp/sandbox:
$ mount | grep sandbox
proc on /tmp/sandbox/proc type proc (rw,relatime)
udev on /tmp/sandbox/dev type devtmpfs (rw,relatime,size=1434720k,...)
/dev/mapper/ubuntu--vg-ubuntu--lv on /tmp/sandbox/realroot type ext4 (rw,relatime)

# That sucks, since it means that even if we were to start a containerized process
# within the sandbox, it would still be able to see the contents of the mountpoints.
# Let's unmount all the mountpoints from the sandbox from the Host's shell:
$ for dir in dev proc realroot; do sudo umount /tmp/sandbox/$dir; done

# In order to actually isolate the process, we need to exit the chroot and
# run the chroot in an actual mount namespace using "unshare -m".
$ sudo unshare -m chroot /tmp/sandbox /bin/bash
[C] $ mount -t proc proc /proc
[C] $ lsns -t mnt
        NS TYPE NPROCS   PID USER             COMMAND
# NOTE: different mount namespace than the host:
4026532328 mnt       2 55221 root             /bin/bash

# Let's create some random bind mounts in the chroot:
[C] $ mkdir -p /tmp/chain{0,1,2}; echo "Start of the chain." > /tmp/chain0/file.txt
[C] $ mount --bind /tmp/chain0 /tmp/chain1; mount --bind /tmp/chain1 /tmp/chain2
[C] $ cat /tmp/chain2/file.txt
Start of the chain.
# We just created a bind-mount chain, nice! Let's check out the mounts with "mount":
[C] $ mount
proc on /proc type proc (rw,relatime)
# Funnily enough, the kernel still "leaks" the name of the real root device.
/dev/mapper/ubuntu--vg-ubuntu--lv on /tmp/chain1 type ext4 (rw,relatime)
/dev/mapper/ubuntu--vg-ubuntu--lv on /tmp/chain2 type ext4 (rw,relatime)

# Because we did NOT also create a new user namespace with "-U", we can mount it:
[C] $ mount -t devtmpfs udev /dev; mount dev/mapper/ubuntu--vg-ubuntu--lv /realroot/
[C] $ mount
...
/dev/dm-0 on /realroot type ext4 (rw,relatime)

[C] $ cat /realroot/root.txt
This file is in the REAL root.

# Now the funky part: open a shell on the Host and check all these dirs:
$ ls /tmp/sandbox/{dev,realroot,tmp/chain{1,2}}
<all dirs should be empty, since they are only mounted in the namespace>
```

Listing 4: Mount namespace showcase.

2.4 Process (PID) Namespaces.

In order to offer a containerized process the same inter-process-related privileges (e.g. `fork`-ing into a new process, **sending signals** to other processes, etc...) which a normal Host system process has, we would need Kernel-level mechanisms to guarantee isolation between processes. This features are referred to as **process (PID) namespace**, and **IPC namespaces**, which you can think of as a little process *sandbox* that our containerized processes can feel free to act real ill in.

```
# Recall that just "chroot"-ing into our sandbox will NOT provide any process isolation:
$ sudo chroot /tmp/sandbox /bin/bash
[C] $ mount -t proc proc /proc
[C] $ ps aux
<shows ALL system processes>

# This is because the Linux kernel will still just show the same "/proc" files
# in the container as seen on the Host. Let's check the Process namespace:
[C] $ lsns -t pid
        NS TYPE NPROCS PID USER COMMAND
4026531836 pid     115   1 root /sbin/init

# Yep, we can clearly see that our "chroot" command is in the same process
# namespace as "/sbin/init" which is the first (and only!) process the Host Linux
# kernel explicitly started when our Host booted. On modern Linux distros, it's systemd:
$ file /sbin/init
/sbin/init: symbolic link to /lib/systemd/systemd

# Here's a dumb idea: what if we tried to run a whole separate systemd process
# within the chroot? That would basically be like having full new Machine,
# since systemd should then start all system services like sshd and others!
[C] $ /sbin/init
Running in chroot, ignoring request.

# Damn, systemd figured out it was in a process namespace which already has another
# systemd running. (recall that running "ps aux" will list the Host's systemd)
# To run our chroot in its own PID namespace, we can pass the "-p" flag to "unshare":
$ sudo unshare --mount-proc -f -p -r chroot /tmp/sandbox /bin/bash
# NOTE: this will mount a "fake" /proc just for our PID namespace:
[C] $ mount -t proc proc /proc
[C] $ ps aux
<only shows the handful of processes in the namespace>
[C] $ lsns -t pid
        NS TYPE NPROCS PID USER COMMAND
# NOTE: different PID namespace:
4026532165 pid       2   1 root /bin/bash

# Without going into too much detail now, we will still NOT be able to run
# systemd in a separate PID namespace without some more setup steps, but
# as we'll be seeing later, it is absolutely possible to do so.
# The resulting container would be called a "machine container", since it
# would contain all the processes (init included) a normal machine would.
$ /sbin/init
Couldn t find an alternative telinit implementation to spawn.
```

Listing 5: Process (PID) and IPC namespace showcase.

2.5 CPU/RAM limiting through cgroups.

As shown in the previous section, we can compose mount and process namespaces to basically boot a whole machine's worth of processes (init process included) in an isolated fashion. However, what is to stop a badly-written C program in the container from leaking infinite RAM? In order to prevent such a scenario, we'd need a mechanism to **C**ontrol system resources for a **group** of process: **cgroups**.

```
# First off, it's useful to note that even the Host system processes are
# running in their own cgroup, which is the "root" of the "cgroup tree":
$ cat /proc/cgroups
#subsys_name    hierarchy    num_cgroups    enabled
cpuset          0            98             1
cpu             0            98             1
blkio           0            98             1
memory          0            98             1
...

# To create and manipulate cgroups, we'll need to install some dedicated tools:
$ sudo apt install -y cgroup-tools
$ sudo cgget -a  # this will show all the properties of our default cgroup (named "")
:   # NOTE: the default root cgroup's name is empty string. ("")
cpuset.cpus.effective: 0-1
...

# Now let's create a cgroup which can only ever have 50MB of RAM:
$ sudo cgcreate -g memory:50mbclub
$ sudo cgset -r memory.max=50M -r memory.swap.max=0M 50mbclub
$ sudo cgget 50mbclub
50mbclub:
cpuset.cpus.partition: member
...
memory.max: 52428800
memory.swap.max: 0
...

# Now let's try to run the chroot in the new cgroup:
$ sudo mount -t devtmpfs udev /tmp/sandbox/dev  # NOTE: mount /dev in advance!
$ sudo cgexec -g memory:50mbclub unshare --mount-proc -C -i -f -p -r chroot /tmp/sandbox
$ mount -t proc proc /proc
$ lsns -t cgroup
# NOTE: because we also passed "-C" to "unshare", we also have our own cgroup namespace!
# This means we can freely create sub-cgroups within our container if we wanted to...
4026532192 cgroup      2   1 root /bin/bash

# Now let's try the resource limiting with a simple read from "/dev/urandom":
$ VAR="$(dd if=/dev/urandom bs=1000000 count=10)"
# 10 MB worked fine:
10000000 bytes (10 MB, 9.5 MiB) copied, 0.618423 s, 16.2 MB/s
# 51 MB will get our process instantly killed for daring to ask for more than 50MB:
$ VAR="$(dd if=/dev/urandom bs=1000000 count=51)"
Killed
```

Listing 6: cgroups showcase.

2.6 Network Namespaces.

With the combined powers of mount, process, and cgroup namespaces, we have basically achieved the ability to run *all* of the processes you would find on your average real-world machine. (from `/sbin/init` to actual user services)

The only missing link is allowing our containerized processes to access the network. This can be simply done by exposing the host's networking stack (L2-L4) to the containers directly (knows as **host networking mode**), but this would mean the container would be able to access Host services through the loopback interface (e.g. `curl http://localhost:80/no_containers_allowed`), as well as manipulate the IP settings of the host itself.

In order permit the full range of networking operations (e.g. assigning an IP to a network interface, using an alternate DNS server, and so on), we'd need to leverage a Linux kernel feature known as **network namespaces**.

```
# We can use our old friend the "ip" command to manipulate network namespaces.
# You can think of network namespaces as the Linux kernel "emulating" the
# networking stack between Layers 2 and 4 of the OSI model, called the "data"
# and the "trasmission" layers, respectively. Let's create a new net namespace:
$ sudo ip netns add ns1
$ sudo ip netns
ns1

# We can execute commands within the namespace with "ip" too, let's "ip a" in it:
$ sudo ip netns exec ns1 ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

# Because this is a fresh network namespace, the only network interface present is
# the loopback ("lo") interface, which is initially DOWN, so let's cheer it UP:
$ sudo ip netns exec ns1 ip link set lo up
$ sudo ip netns exec ns1 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo

# Now it's UP, we can have our Host ping itself within the namespace:
$ sudo ip netns exec ns1 ping 127.0.0.1
<ping should work, since it's pinging itself after all...>

# As a fun fact, the namespaces are no more than empty files located here:
$ file /var/run/netns/ns1
/var/run/netns/ns1: empty
```

Listing 7: Network namespaces basics.

```
# Only having a loopback in the namespace isn't very useful, we'd ideally
# like to have a Host-side AND container-side L2 interface which are linked
# so that the container can reach the host on an L2 level and vice-versa:
$ sudo ip link add veth0 type veth peer name veth1
$ ip a
# We now have a veth0 we'll configure on the host, and veth1 for the namespace, both linked:
4: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT ...
    link/ether ee:8e:d2:e5:60:02 brd ff:ff:ff:ff:ff:ff
5: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT ...
    link/ether aa:48:1e:26:f8:33 brd ff:ff:ff:ff:ff:ff

# Let's now "move" veth1 into the network namespace, and configure them with random IPs:
# This will set the "veth1" NIC's network namespace to be "ns1".
$ sudo ip link set veth1 netns ns1
$ ip a
<we will now see that veth1 has "disappeared" from the host network namespace>

# Let's add an IP to veth0 on the Host and bring it up:
$ sudo ip addr add 192.168.200.1/24 dev veth0
$ sudo ip link set dev veth0 up
# And now let's configure "veth1" within the network namespace:
$ sudo ip netns exec ns1 ip addr add 192.168.200.2/24 dev veth1
$ sudo ip netns exec ns1 ip link set dev veth1 up
# Pinging the host from the container NS and vice-versa should both work now:
$ ping 192.168.200.2; sudo ip netns exec ns1 ping 192.168.200.1
<should work>

# That's cool and all, but "ip netns exec" only executes commands within network namespaces,
# so in order to combine it with mount and PID namespaces, we need "unshare":
$ sudo ip netns exec ns1 unshare --mount-proc -i -f -p -r chroot /tmp/sandbox /bin/bash
[C] $ ip a
<we can now see the appropriate interfaces inside our "container">

# As you can imagine, allowing this container to access the Internet would now
# be as easy as setting some IPv4 forwarding rules on the Host using "iptables".
# NOTE: replace "eth0" with the name of your Host's external network interface.
$ sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
$ sudo iptables -A FORWARD -i veth0 -o eth0 -j ACCEPT
$ sudo iptables -A FORWARD -i eth0 -o veth0 -m state --state RELATED,ESTABLISHED -j ACCEPT

# NOTE: since we used "unshare -r", we'll need to add the default route from the host:
$ sudo ip netns exec ns1 ip route add default via 192.168.200.1

# We'll also need to set our own DNS server in the container:
[C] $ rm /etc/resolv.conf; echo "nameserver 8.8.8.8" >> /etc/resolv.conf

# The containerized shell should be able to access the Internet now:
[C] $ nslookup google.com
<DNS should work>
[C] $ wget google.com
<should download index.html>
```

Listing 8: Network namespaces showcase: adding virtual NICs to Namespaces.

2.7 Namespaces and `unshare` recap.

As mentioned, namespaces are Linux kernel features, so the Kernel is the only entity which manages the namespaces' existence, what processes are part of which namespace, and what implications each namespace has.

Nobody actually "creates namespaces", instead programs (e.g. "unshare") will ask the kernel to run an executable (e.g /bin/bash) in a new namespace. The kernel will automatically create the namespace internally if it doesn't exist, or it will add the new process in the namespace if there happens to already be a process within it.

```
# Without much ado, let's use "unshare" to set up ALL the namespaces:
# -f: will allow the child process to fork() its own child processes.
# -m: create new mount namespace.
# -p: new process namespace.
# -i: new IPC namespace.
# -n: new network namespace.
# --mount-proc: allow container to mount /proc.
# -U: new user namespace.
# -r: create a host-container user mapping for "root".
# -c: create a host-container User mapping for the current user. ("root" when "sudo"-ing)
# -C: new cgroup namespace: allows defining sub-cgroups in the container.
# -u: new Unix Time Sharing (UTS) namespace: allows hostnames.
# -T: new Time namespace: allows different time settings in container.
# -R: automatically chroot into /tmp/sandbox.
$ sudo unshare -f -m -p -i -n --mount-proc -U -u -r -c -C -R /tmp/sandbox bash

# But how are these actually defined? Let's start a "sleep" process we can poke around:
$ sudo unshare -f -m -p -i -n --mount-proc -U -u -r -c -C -R /tmp/sandbox sleep inf

# Looking at the process in /proc, we can see it has a "ns" directory.
# /proc/$PID/ns actually defines some symlinks which, instead of linking to
# a file, actually just contain a special string denoting namespace IDs:
$ sudo ls -l /proc/$(pgrep sleep)/ns
lrwxrwxrwx 1 root root 0 Jun  5 12:05 mnt -> 'mnt:[4026532267]'
lrwxrwxrwx 1 root root 0 Jun  5 12:05 net -> 'net:[4026531840]'
lrwxrwxrwx 1 root root 0 Jun  5 12:05 pid -> 'pid:[4026532270]'
lrwxrwxrwx 1 root root 0 Jun  5 12:05 user -> 'user:[4026532266]'
...

# Note the magical-looking "NAMESPACE_TYPE:[NAMESPACE_ID]" notation.
# These namespaces were automatically created when "unshare" asked Linux
# to run "sleep inf" inside different namespaces.
# These are the exact IDs that the "lsns" command reads and show us.

# Additionally, the "uid_map" and "gid_map" files encode how system users
# like the real root (UID 0) map inside the container:
$ cat /proc/$(pgrep sleep)/uid_map
         0          0          1

# We can use the "nsenter" utility to execute a command in our namespaces:
# E.g. "nsenter -n" will exeute the "ip a" command in the same network namespace:
$ sudo nsenter -t $(pgrep sleep) -n ip a
```

Listing 9: Namespaces recap.

Chapter 3

Container Runtimes: one-command containers.

As we've seen in Chapter 2, Containers are nothing more than normal processes with extra steps...quite a lot of annoying repetetive extra steps...

Naturally, a class of programs known as **Container Runtimes** emerged to fill this niche: they run the process you tell it in the chroot directory you give it with all the namespacing and security setup you could want being set up almost entirely automatically by the Runtime.

3.1 Machine vs. Process vs. Micro-VM Containers

Depending on how and what we start as the first process (PID 1), Containers and their Runtimes can be broadly categorized as follows:

- **Process Containers** (aka Application Containers): where the user explicitly defines the path to a single executable application (e.g. `/usr/bin/nginx`) which should be running in the Container. If the process exits at any stage, the process Container is considered dead.
- **Machine Containers**: (aka System Containers): where the Runtime will always start `/sbin/init` in the container, which will then start the full selection of a system services a real Machine would have. These Containers very much act like Virtual Machines, allowing you to freely install and start any number of services just like on any full system.
- **Micro-VM Containers**: some Container Runtimes can actually use QEMU or some other full virtualization solution to run each Container (be it a Process or Machine Container) in its own miniature VM. This provides added security while still allowing to deploy the same Container images than non-VM Runtimes can.

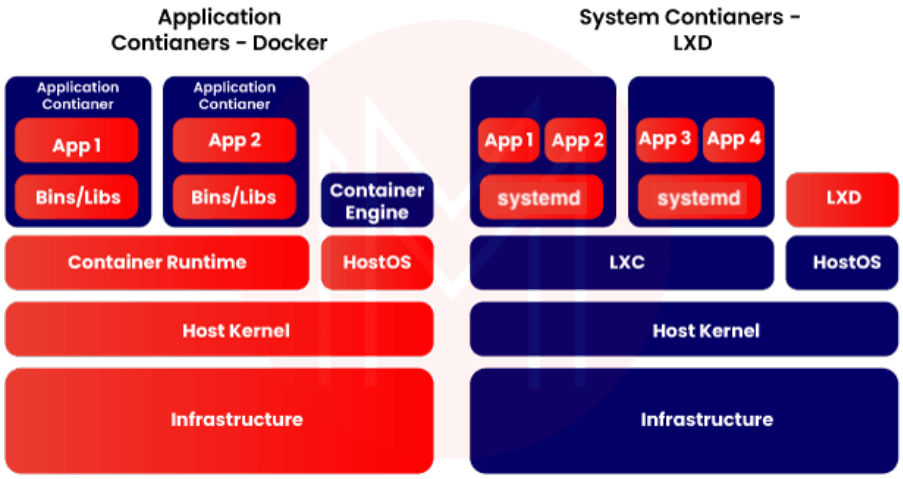

Figure 3.1: Process vs. Machine Containers. (Docker vs LXD)

3.2 LXC: Linux Containers Distilled.

In this section we will mess around with **LXC**, one of the oldest and most mature (yet quite bare-bones in terms of features) Machine Container Runtimes around. As we'll soon discover, LXC is really nothing more than the composition of the various namespacing and isolation features we discussed in Chapter 2.

```
# First off, let's install LXC and some Container templates for it:
$ sudo apt install -y lxc-utils lxc-templates

# After installing it, we can see that it's made up of 3 services:
$ systemctl status lxc*
# NOTE: "lxc" and "lxc-net" are "Type=oneshot" systemd services which are meant
# to execute once to set some things up, then exit. (hence "active (exited)")
* lxc.service - LXC Container Initialization and Autoboot Code
     Loaded: loaded (/lib/systemd/system/lxc.service; enabled; vendor preset: enabled)
     Active: active (exited) since Fri 2023-06-02 12:19:15 UTC; 2 days ago
* lxc-net.service - LXC network bridge setup
     Loaded: loaded (/lib/systemd/system/lxc-net.service; enabled; vendor preset: enabled)
     Active: active (exited) since Fri 2023-06-02 12:19:15 UTC; 2 days ago
# NOTE: "lxcfs" is responsible for setting up some special files in running containers
# such as patching "/proc/uptime" or "/sys/devices" to reflect the Container's setup.
* lxcfs.service - FUSE filesystem for LXC
     Loaded: loaded (/lib/systemd/system/lxcfs.service; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2023-06-02 12:19:14 UTC; 2 days ago
     ...
# NOTE: "lxc-monitord" is a Daemon service which simply monitors and restarts
# any containers which are configured to be always on.
* lxc-monitord.service - LXC Container Monitoring Daemon
     Loaded: loaded (/lib/systemd/system/lxc-monitord.service; enabled)
     Active: active (running) since Fri 2023-06-02 12:19:14 UTC; 2 days ago
     ...

# Note the fact that NONE of these services actually run containers themselves, all
# of that will be set up by the actual container creation/starting/etc commands.

# Let's set up an Ubuntu Container to cherish as a pet forever:
$ sudo lxc-create -t ubuntu -n PetContainer
Checking cache download in /var/cache/lxc/jammy/rootfs-amd64 ...
Installing packages in template: apt-transport-https,ssh,vim,language-pack-en
Downloading ubuntu jammy minimal ...
I: Target architecture can be executed
I: Retrieving InRelease
...

# You will note that the output looks suspiciously like "apt install"'s output.
# That's because LXC does Machine Containers, so it basically installs a whole
# new Ubuntu system (minus the Kernel), which can take a while...
# During this time, you can periodically grep for the "apt" activity in the container:
$ ps aux | grep apt
# This "apt dist-upgrade" is running inside the Container while it's installing.
root       43696  7.0  5.3  77492 66280 pts/1    S+   10:24   0:00 apt-get dist-upgrade -y
```

Listing 10: LXC setup.

```
# After running the "lxc-create" command we can check if our PetContainer is defined:
$ sudo lxc-ls -f
NAME         STATE   AUTOSTART GROUPS IPV4 IPV6 UNPRIVILEGED
PetContainer STOPPED 0         -      -    -    false

# The "-f" stands for "fancy" btw, no-joke, check "man lxc-ls".

# Let's start our fancy new pet:
$ sudo lxc-start PetContainer
$ sudo lxc-ls -f
NAME         STATE   AUTOSTART GROUPS IPV4       IPV6 UNPRIVILEGED
PetContainer RUNNING 0         -      10.0.3.140 -    false

# We can "attach" our console to the container's console anytime using:
$ sudo lxc-attach PetContainer
[C] $ ps aux
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      1  0.0  0.8  99796 10596 ?        Ss   10:31   0:00 /sbin/init
root     37  0.0  0.9  31328 10940 ?        S<s  10:31   0:00 /lib/systemd/systemd-journald
<small list of processes running in the container>

# Because this is a Machine Container, it can install/run any number of services:
[C] $ apt install -y openssh-server
[C] $ systemctl status sshd
* ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2023-06-05 10:31:36 UTC; 4min 44s ago

# We even have our own IP address in a virtual private subnet just for containers!
[C] $ ip a
2: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP qlen 1000
    link/ether 00:16:3e:26:a1:36 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.3.140/24 metric 100 brd 10.0.3.255 scope global dynamic eth0

# You can even "ssh ubuntu@<container_IP>" from the host! (default password is "ubuntu")
# External networking should also work just fine, as LXC set up NAT forwarding rules:
[C] $ ping google.com
\PING google.com (142.250.201.206) 56(84) bytes of data.
64 bytes from bud02s35-in-f14.1e100.net (142.250.201.206): icmp_seq=1 ttl=56 time=9.25 ms

# And we can, of course, create and manage files freely:
[C] $ echo "LXC is the crossover between Linux and Cool." > /lxc.txt

# Also, we can take note of the unique namespaces created for this container:
[C] $ lsns
        NS TYPE   NPROCS PID USER            COMMAND
4026531834 time       17   1 root            /sbin/init
4026531837 user       17   1 root            /sbin/init
4026532162 mnt        14   1 root            /sbin/init
4026532165 ipc        17   1 root            /sbin/init
4026532166 pid        17   1 root            /sbin/init
4026532167 cgroup     17   1 root            /sbin/init
4026532168 net        17   1 root            /sbin/init
```

Listing 11: LXC introduction.

```
# First off, let's check out what that "lxc-templates" package installed:
$ dpkg-query -L lxc-templates
/usr/share/lxc/config/ubuntu.common.conf
/usr/share/lxc/config/ubuntu.lucid.conf
/usr/share/lxc/config/ubuntu.userns.conf
...
/usr/share/lxc/templates/lxc-ubuntu

# There's a lot there, but let's first check out the template files:
$ file /usr/share/lxc/templates/lxc-ubuntu
/usr/share/lxc/templates/lxc-ubuntu: Bourne-Again shell script, text executable

# Oh wow, it's a script which just install the system in a directory!
# Most notable is the use of "debootsrap" (which installs Debian packages just
# like "apt", but in a different directory of your choosing instead of "/")
$ grep debootstrap /usr/share/lxc/templates/lxc-ubuntu
...
debootstrap --verbose $debootstrap_parameters --components=main,universe --arch=$arch

# In order to see how LXC works behind the scenes, "lxc-attach" and run:
$ sudo lxc-attach PetContainer
[C] $ sleep inf

# Now we can easily search for that sleep process ON THE HOST and check out its settings:
$ sudo ls -l /proc/$(pgrep sleep)/ns
lrwxrwxrwx 1 root root 0 Jun  5 10:55 cgroup -> 'cgroup:[4026532167]'
lrwxrwxrwx 1 root root 0 Jun  5 10:55 ipc -> 'ipc:[4026532165]'
lrwxrwxrwx 1 root root 0 Jun  5 10:55 mnt -> 'mnt:[4026532162]'
...

# NOTE: we didn't set any namespace settings on our "sleep" process, it just automatically
# inherited them from the container's "/sbin/init", which had them set by "lxc-start".

# And we can use our good friend "nsenter" to join the container namespace:
$ sudo nsenter -t $(pgrep sleep) -n ip route
<will show you the IP confinguration in the Network Namespace of the container>

# The "ps" executable reads /proc, so to run it we must enter both the
# process ("-p") namespace, as well as the mount ("-m") namespace:
$ sudo nsenter -t $(pgrep sleep) -m -p ps aux
<will show the processes within the Process Namespace of the container>

# And even query the "systemd" Daemon inside the container if entering both
# the mount ("-m") and process ("-p") namespaces:
$ sudo nsenter -t $(pgrep sleep) -m -p systemctl status
<will show status of services inside the container>

# Lastly, we should not be surprised at all to find a directory with the Container:
$ ls /var/lib/lxc/PetContainer
# There's a config file will the container's settings:
$ cat /var/lib/lxc/PetContainer/config
# And a directory with the container's root FS:
$ cat /var/lib/lxc/PetContainer/rootfs/lxc.txt
LXC is the crossover between Linux and Cool.
```

Listing 12: LXC inner workings.

Chapter 4

The Open Container Initiative (OCI) and the drive for standardization.

The **O**pen **C**ontainer **I**nitiative (or **OCI**) is an industry-wide effort to formalize the way Containers are defined, run, and shipped across environments. The effort was initially spearheaded by the Docker team in 2015, and later adopted by the Linux Foundation.

It can not be understated how crucial the standardization efforts of the Linux Foundation have been in maintaining the direction of the industry and preventing a Unix Wars-style fracture in Cloud Computing as a whole during the early days of Containerization on Linux.

To better appreciate this, let's take a moment to look at a map of CNCF projects, which are Cloud Computing and Container-related tools under the umbrella of the **C**loud **N**ative **C**omputing **F**oundation (**CNCF**), another Linux Foundation effort.

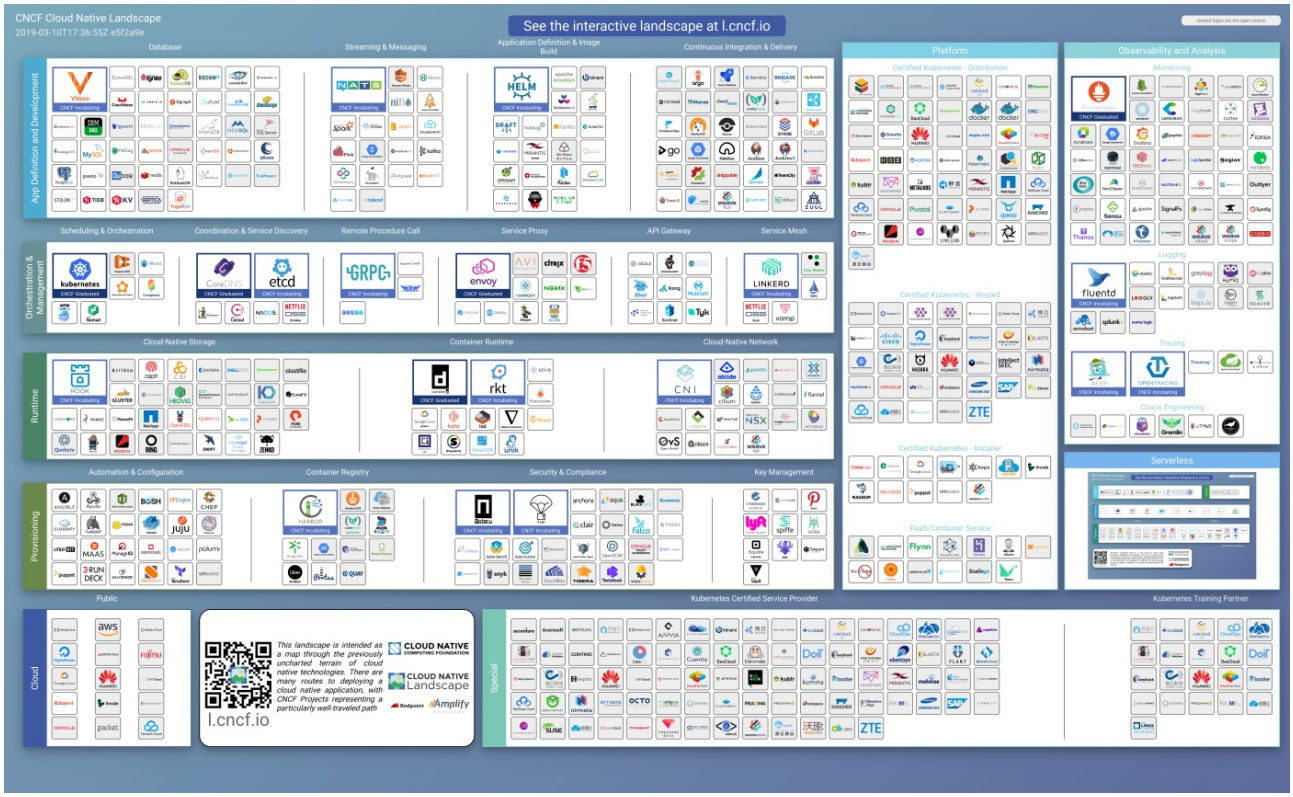

Figure 4.1: CNCF Project Landscape 2019.

All of the little boxes in Image 4 are individual projects which consume, interact with, or are consumed by some of the other projects to somehow result in a coherent containerization ecosystem where any specific user problem has a specialized box with the solution.

4.1 Runtimes: the best of times.

As mentioned, the early stages of Containerization on Linux saw the rise and fall of numerous projects, most notable of which being the **Container Runtimes** which saw an explosion and eventual settling down over the years. Let's go through some of that timeline now.

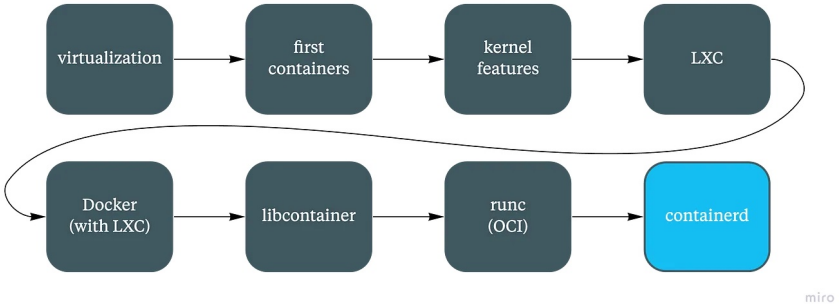

Figure 4.2: Container Runtimes Timeline.

- **BSD jails** (1999): even as early as the 90s, the idea of containerization (or "OS-level virtualization") was implemented in FreeBSD through the concept of "jails". It was, however, a considerably more "monolithic" solution than modern Linux containers. and as such, they were considerably less flexible from the application's point of view.
- **Solaris Zones** (2005): a more formal and flexible OS-level virtualization solution called "Zones" made its debut in Solaris. It even offered ABI translation layers so you could run Linux binaries in special types of Solaris zones, and many more insanely cool features. Try OpenIndiana!
- **LXC** (2008): one of the earliest implementations of a Container Runtime for **Linux Containers**. It was basically just the composition of all the Linux kernel's namespacing features discussed in Chapter 2 into an easy to set up and use, but quite minimal Container Runtime system.
- **LXD** (2013): a service built on top LXC which offers a nice API and more user-friendliness with the added ability of creating and managing Containers in Micro Virtual Machines too.
- **Docker 1.0.0's Runtime** (2013): Docker is an all-in-one Container Engine and Manager whose initial Runtime implementation leveraged LXC.
- **rkt** (2014): "Rocket" was a container runtime developed for a container-focused Linux distribution named CoreOS. CoreOS was acquired in its heyday and is basically abandoned now.
- **runc** (2015): a Runtime conceived to act as a Reference Implementation for Container Runtimes moving foward. It was created by the **O**pen **C**ontainer **I**nitiative (**OCI**) in hopes of standardizing the now-unwieldy ecosystem that had emerged in the Container industry.
- **CRI-O**: a runtime specifically implemented to be used by Kubernetes, a container orchestrator which can manage multiple Container Runtimes on multiple servers through a specification called the **C**ontainer **R**untime **I**nterface. (or CRI)
- **gVisor** (2018): a Runtime developed by Google which re-implements parts of the Linux system in a half-Micro-VM, half-Process Container implementation meant to offer increased security over the standard Linux namespacing features.
- **Kata** (2018): a Runtime which starts a small Virtual Machine using hardware-assisted virtualization (similar to KVM VMs) whose sole purpose is to run the process(es) in the containers in as secure and isolated a fashion as fully-fledged Virtual Machines.
- **containerd** (2015): a "Container Runtime Manager" which provides a single interface for managing different containers on different runtimes, so you could run multiple container types (e.g. both `runc` and `kata` containers) on the same server. `containerd` is actually what the modern Docker Engine, as well as most Kubernetes installations use behind the scenes.

4.2 Docker and Container Engines: more than just their Runtime.

Runtimes are nice for *running* a Containerized process in the here and now, but who's *monitoring* the process? Will anyone ever know if the process dies? And where did we get all these Container images anyway!?

Running a containerized process is only half the battle, and something needs to be put in charge of the Runtime to manage the containers and *drive* things forward like the...*engine* of a car?

In the modern CNCF Landscape, **Container Engines** like Docker are the preferred all-in-one solution for building, distributing, and running Containers, and any Container Engine worth its salt should offer the following features expected by its users:

- define an **image format**: before we can ask the service to run containers for us, we first need to define how we're going to give it the container. This implies the need to define a format for container images which should contain:
  - **all the files for the container**, including the main executable and any libraries or configuration/data files it may need.
  - an **entrypoint** for the container: which binary command from the container should be executed when the user runs the container.
  - **additional container metadata**: random things like what username to start the container as (in case your app expects a specific one), what timezone to set for it, and most of everything else you could imagine a process might need.
- implement or use a **container Runtime**: the program(s) which actually create all the namespaces and set up our containerized process to run within them.
- offer an **API**: in order to actually offer the containerization services we added to our solution to our users, we'd need to define a way for users to interact with the Container Engine. This is most easily done by offering our users an HTTP-based REST API.
- offer **client tools** to our service: our users will probably not want to use our REST API through `curl` directly, so we should offer them a friendly Graphical UI (e.g. Docker Desktop) or a Command Line Interface (the `docker` command) for them.

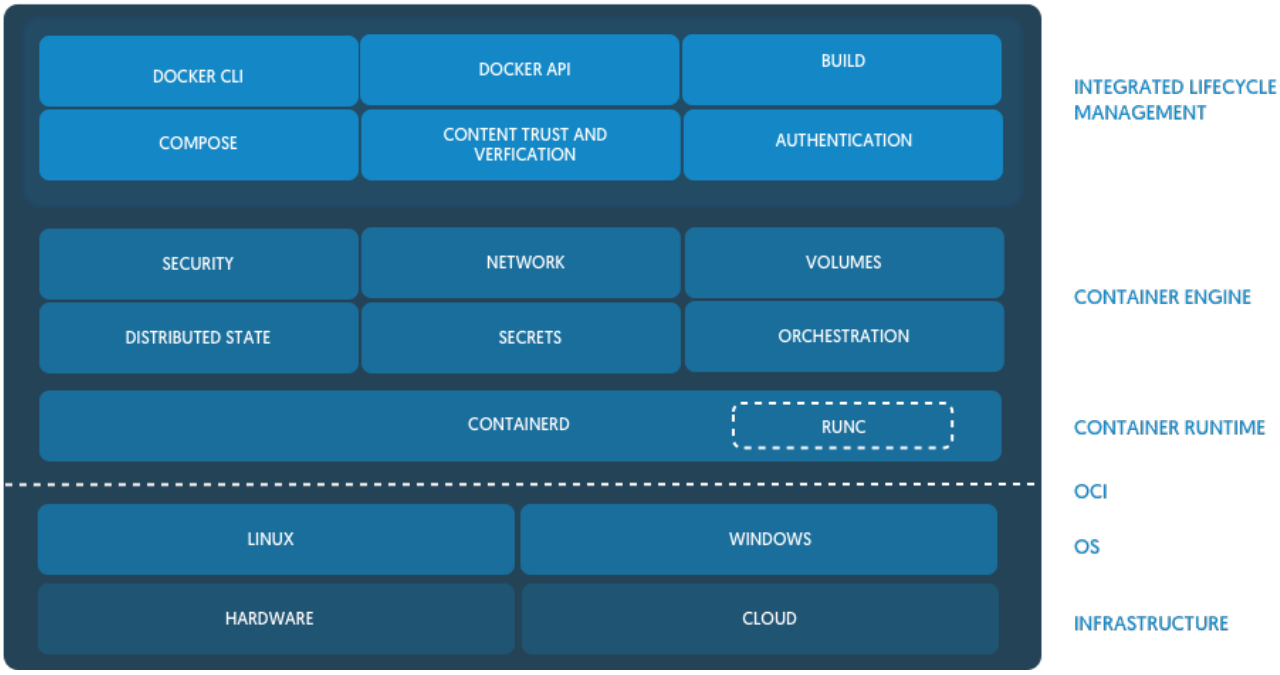

Figure 4.3: Docker Container Engine Components.

4.3 Container Images: self-contained worlds.

Arguably the most important thing to determine before running your app in a Container is defining how it will be "packaged" and distributed.

At the end of the day, a containerized application will just run as a normal system process, so it will need the same things any other service or app on your system needs:

- the **executable's code** itself (d'oh!): for binaries that is obviously just the compiled ELF/EXE file with all the CPU instructions in it, but for interpreted (aka scripting) languages like Bash or Python, we'll need the actual interpreter (i.e. `/usr/bin/{bash,python,etc}`) as well.
- any **required libraries**: the vast majority of executables rely on library calls (e.g. `printf` and any other functions you call when you `#include<stdio.h>`) to eliminate code duplication. Where this library code is located depends on how the binary was compiled:
  - **statically linked** binaries: when statically compiling programs, the resulting executable file will contain any library code within the actual ELF/EXE. This hypothetically makes the binary work on any kernel with a compatible **A**pplication **B**inary **I**nterface. (ABI)
  - **dynamically linked** binaries: these binaries are compiled *without* any of the library code it relies on, and instead will dynamically load and call libraries from the system. These libraries are the various "shared object" files found in `/lib`, `/usr/lib`, etc, and are usually configured through a mechanism known as `ldconfig`.
- **configuration files**: most serious apps and services offer so many options that you usually need to give them a configuration file on startup. (i.e. all the files we've been seeing in `/etc`)
- **application data files**: apps which "carry state" will inevitably want to save that state somehow, most times by saving it in a file on disk.

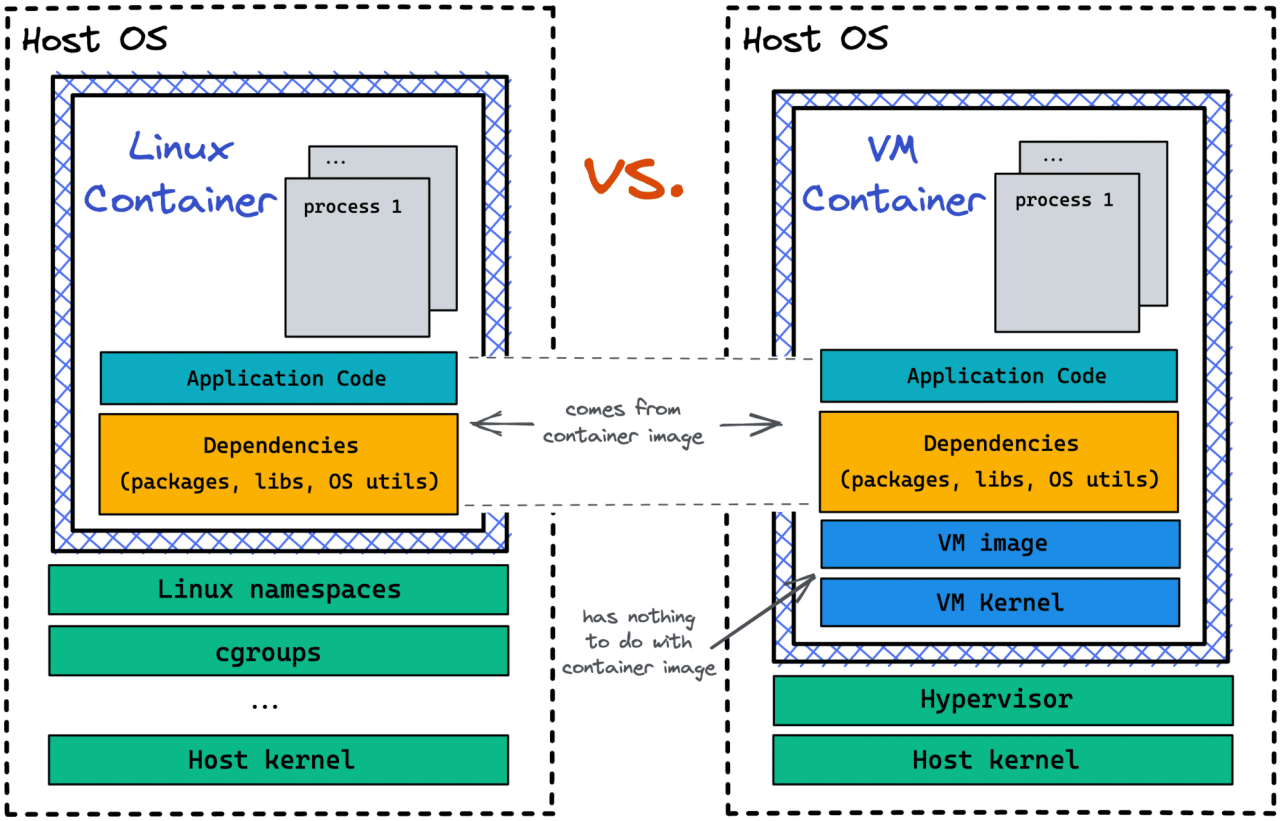

Figure 4.4: Where the Container Image comes in for Process (left) vs Micro-VM Containers.

4.4 Container Image Formats: a way of sharing self-contained worlds.

All in all, it is clear that any **Container image format** will need to basically offer a full filesystem to the containerized application as we've seen in 4.3. In this sense, if we were asked to invent a Container image format, we can imagine a number of solutions for packaging said filesystem with our application:

- compressing it into an **archive**: a naive approach for sure, but it absolutely works. All we'd need to do is unpack the archive in a directory and start a containerized process within a `chroot`.
- using **vitual disk image formats**: riding on the back of years of developments in the virtualization industry, we could also just use a disk image format which we mount and run the containerized process in it. A lot of the more mature image formats like QEMU's QCOW even offer advanced features like **C**opy **O**n **W**rite (COW) which would even enable us to "snapshot the container" as if it were a VM.
- **something better**: we could, of course, put in the work and define a new specialized image format for containers which combines the speed and compression of an archive with the advanced features of a disk image. This is usually achieved by "layering" files from multiple smaller archives through a so-called `union filesystem` like UnionFS or OverlayFS.

4.4.1 The OCI Container Image Format.

While we will be inspecting the exact structure of OCI image spec later in this document, it has the following hierarchical structure:

- **an image name or identifier** (d'oh!) of the form `NAME:VERSION` (e.g. `ubuntu:22.04`) which actually points to a Manifest List.
- the **Manifest List**, which is a list of individual **Image Manifests** to use on different CPU architecture and OSes. (e.g. `linux/amd64`)
- the **Image Manifest**: an actual archive (most often a `.tgz`) which contains the individual Filesystem Layers of the Container Image and their configurations.

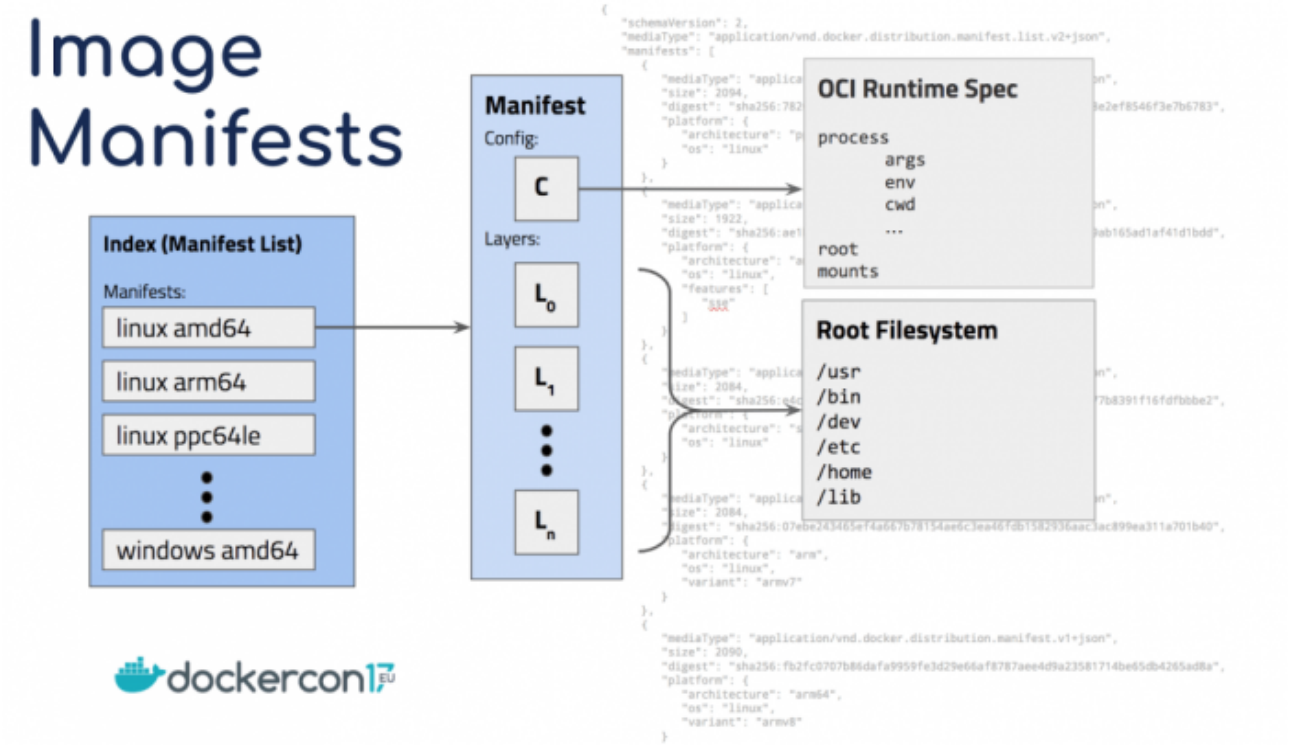

Figure 4.5: OCI Image Manifest Structure.

Chapter 5

Working with Docker.

Docker is the de facto **Container Engine** (4.2) implementation of the world. It offers a robust HTTP-based REST API which users can access throgh an assortment of both command line and GUI-based interfaces to build, run, and manage **Process Containers**. (3.1)

The term "Docker" is usually used as an umbrella term for a "LEGO set of toolkit components" for building and running Containers, and is actually comprised of a handful of components:

- **Docker Client**: the method users interact with the Docker host. (e.g. the `docker` command, or the Docker Desktop application)
- **Docker Host**: the actual server running `dockerd` (the Docker Daemon) which does all the actual work. This uses a number of different components like Overlay Filesystems to mount the Container's image, and a **Container Runtime** (3) for running the actual containerized processes.
- **Docker Registry**: a repository of Docker Images for the Host to pull and run containers from. Note that the Docker Host technically has a pseudo-registry which stores local images within it, but it's not accessible to users or other services, so one needs to be deployed separately.

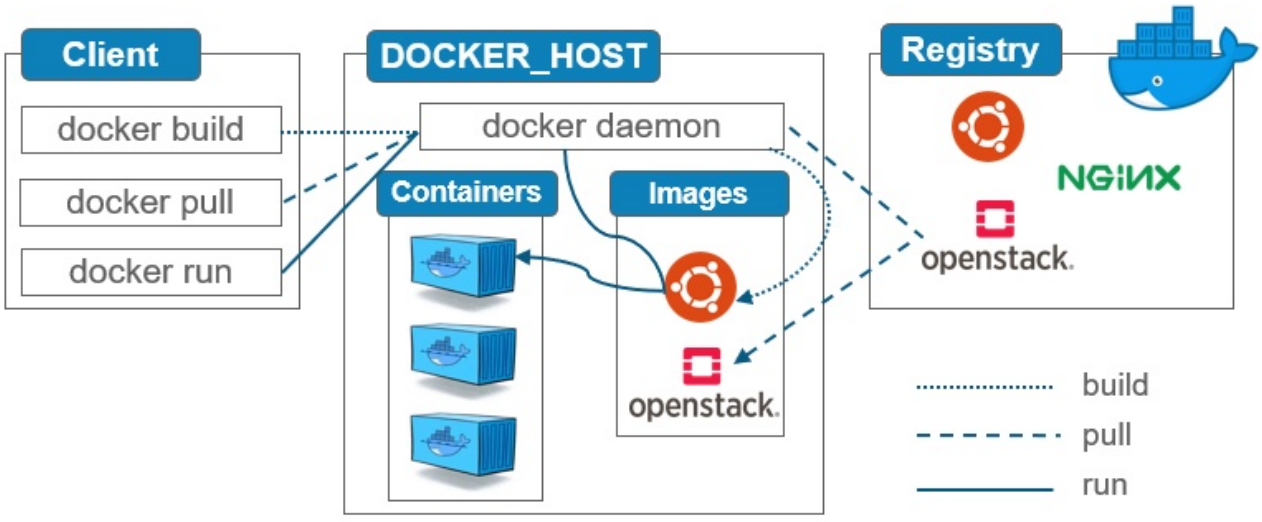

Figure 5.1: Docker Architectural Layer Overview.

5.0.1 Docker Setup and Intro.

```
# On Ubuntu, we can install Docker from both "apt" and "snap", but we'll go with "apt":
$ sudo apt install -y docker.io

# We should now see the Docker Daemon running:
$ sudo systemctl status docker
* docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2023-06-02 12:19:17 UTC; 3 days ago
     ...
     CGroup: /system.slice/docker.service
             -959 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

# Notice that it's just running "/usr/bin/dockerd" with an argument to talk to containerd:
$ systemctl status containerd
* containerd.service - containerd container runtime
     Loaded: loaded (/lib/systemd/system/containerd.service; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2023-06-02 12:19:15 UTC; 3 days ago

# Containerd is the "Runtime Manager"  which Docker asks to run a container.
# In turn, containerd will ask an actual runtime like "runc" to run the container.
$ containerd config default | grep default_runtime_name
# NOTE: the Runtime containerd uses by default is "runc", but it can use others too.
      default_runtime_name = "runc"

# Note the distinct lack of a "docker-registry" service for hosting images.
# We'll need to set up and run our own registry later, but for now we can
# let Docker use DockerHub so we can pull some images and have some fun:
$ sudo docker info | grep Registry
 Registry: https://index.docker.io/v1/

# Let's do what any good computer scientist would do: force it to say Hello to us!
$ sudo docker run ubuntu:22.04 echo "Hey there!"
Hey there!

# Sweet, Docker knew what was best for it and did what we asked:
# - it downloaded (or "pulled") the "ubuntu:22.04" image from DockerHub.
# - it set up all the container filesystem(s) and namespaces.
# - it executed the "echo" in the container, as expected.

# Note that despite "echo" executing successfully, the container is still defined:
$ sudo docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS
17e32da275fd   ubuntu:22.04   "echo 'Hey there!'"   2 seconds ago   Exited (0) 1 second ago

# That's because the "sandbox" created for the container can be re-started anytime!
# Check the ID of your container and force it to give you as much attention as you need:
$ for i in $(seq 10); do sudo docker start -i 17e32da275fd; done
Hey there!
...
```

Listing 13: Docker Setup and Introduction.

5.1 Poking around Docker Containers.

```
# Well, what're we waiting for? Let's start a container and poke around it!
$ docker run --name guinea-pig ubuntu:22.04 bash
<nothing seems to happen at all>
$ docker ps
<not a single container named "guinea-pig" in sight?!>

# What? Has Docker grown a personality so fast? Is it refusing our commands already?!
# Well no, looking deeper, it actually looks like Docker did run it but Bash "exited 0"?!
$  docker ps -a
CONTAINER ID   IMAGE         COMMAND   CREATED         STATUS                     NAMES
480b5714d91b   ubuntu:22.04  "bash"    3 seconds ago   Exited (0) 1 second ago   guinea-pig

# What actually happened is that Docker did indeed create the container and
# executed "bash" in it, Docker does NOT set up "console emulation" by default.
# (aka no virtual terminal like the one you're running this in right now)
# This means Bash had nothing to print to, so it just called it a day and exited 0...
# We need to use the "-t" argument to "docker run/start" for it to set up a TTY.
$ docker run -t --name guinea-pig-2 ubuntu:22.04 bash
<a new shell opened, but the "Enter" key seems to have stopped working>

# Hahhaaa gat'eem. "-t" may set up terminal emulation so Bash has something to
# print to, but that doesn't mean you have any right to interact with it!
# We also needed to add the "-i" argument to tell Docker to set up interactivity.

# There is unfortunately nothing you can do if you ever forget the "-it" in the
# real world, so let this valuable lesson sink in while you close and re-open
# whatever Terminal app you're using so we can try the full command now:
$ docker run -ti --name guinea-pig-3 ubuntu:22.04 bash

# Finally, a shell to mess around in! We can notice the the IDs of the namespaces in it:
[C] $ lsns
        NS TYPE   NPROCS PID USER COMMAND
# The 'time' namespace is identical to the host, but all others should be different:
4026531834 time        2   1 root bash
4026531837 user        2   1 root bash
4026532583 mnt         2   1 root bash
4026532586 pid         2   1 root bash
4026532587 net         2   1 root bash
...

# Just our shell and the stray "ps" command should be the only processes running:
[C] $ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.4  0.1   4620  3700 pts/0    Ss   14:58   0:00 bash
root          10  0.0  0.0   7060  1568 pts/0    R+   14:58   0:00 ps aux

# We can, from this container, also freely access the Internet:
[C] $ ping google.com
<should work>
```

Listing 14: Poking around Docker Containers. (1/3)

```
# Let's create a random file in the container and leave it running:
[C] $ echo "This file is not random at all." > /expected-file.txt

# But where on the Host did this "expected-file.txt" we just saved end up, exactly?
# Let's open a shell on the Host and check the container's settings using "docker inspect":
$ docker inspect guinea-pig-3
<ginormous wall of JSON text with ALL of the container settings will appear>
{
  # Here's the container's unique Docker-wide ID:
  "Id": "b47d91581bb8b646f28b2c03c08748318b05a7ae8b6f3c5b3862295a94138d32",
  # Here's the command used to start the main process of the container:
  "Path": "bash",
  # Here's the ID of the "ubuntu:22.04" image we used to start the container:
  "Image": "sha256:3b418d7b466ac6275a6bfcb0c86fbe4422ff6ea0af444a294f82d3bf5173ce74",
  "State": {
      "Status": "running",
      # NOTE: this is the PID of the containerized "bash" process.
      "Pid": 80830, ... }
  # These fields all point to files Docker created for the container, like /etc/hostname
  "ResolvConfPath": "...", "HostnamePath": "...", "HostsPath": "...",
  # This is the path the output of the containerized process will be saved in:
  "LogPath": "/var/lib/docker/containers/${CONTAINER_ID}/${CONTAINER_ID}-json.log",
  "GraphDriver": {
    # This contains info on the "working directories" Docker merges image layers in:
    "Data": {
        "MergedDir": "/var/lib/docker/overlay2/${SOME_UGLY_ID}/merged",
        "UpperDir": "/var/lib/docker/overlay2/${SOME_UGLY_ID}/diff",
        "WorkDir": "/var/lib/docker/overlay2/${SOME_UGLY_ID}/work"
  }},
  # Config defines general config options for the container:
  "Config": {
    "Hostname": "b47d91581bb8",
    # These are the pesky options we needed the "-i" argument for:
    "AttachStdin": true, "AttachStdout": true, "AttachStderr": true,
  }
  "Mounts": [ contains mountpoint settings... ],
  "NetworkSettings": { contains network settings ... }
  ...

# As we'll see later, container images are "layered", and anything we write in
# our container will be saved in the separate "UpperDir" location:
$ cat /var/lib/docker/overlay2/${SOME_UGLY_ID}/diff/expected-file.txt
This file is not random at all.

# While the container is running, the "pre-assembled layers" the containerized
# processes will be started in will be located where the "WorkDir" points to:
$ ls /var/lib/docker/overlay2/${SOME_UGLY_ID}/work
<the contents of the filesystem of the container, complete with 'expected-file.txt'>

# You can even see the image layers mounted with OverlayFS on your host:
$ mount | grep "${SOME_UGLY_ID}"
overlay on <MergedDir> type overlay (lowerdir=<image>,upperdir=<UpperDir>,workdir=<WorkDir>)
```

Listing 15: Poking around Docker Containers. (2/3)

```
# Remeber that we could see the "Pid" field of the output of "docker inspect"?
$ docker inspect guinea-pig-3 | grep Pid
            "Pid": 80830,

# As mentioned, that is the PID of the "bash" process inside our container:
$ ps -u -p 80830
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       80830  0.0  0.1   4624  3616 pts/0    Ss+  15:12   0:00 bash

# But who actually started that process? Let's check its parent process:
$ ps -u -p $(ps -o ppid= -p 80830)
USER    PID  COMMAND
root  80806  /usr/bin/containerd-shim-runc-v2 -namespace moby -id ${CONTAINER_ID}
             -address /run/containerd/containerd.sock

# Well would you look at that! The parent of our containerized process is a so-called
# "shim" (aka plugin) for containerd which just uses "runc" as the runtime.

# We can use the "ctr" command to interact with containerd and ask what it sees:
$  ctr --namespace moby containers list
CONTAINER                           IMAGE RUNTIME
<the ID of your container>             - io.containerd.runc.v2

# Here's even more JSON for you, this is how the container looks to containerd:
$ ctr --namespace moby containers info ${CONTAINER_ID}
{
  # The Docker ID we were seeing was actually generated by containerd all along!
  "ID": "b47d91581bb8b646f28b2c03c08748318b05a7ae8b6f3c5b3862295a94138d32",
  # Because Docker was the one that handled unpacking the image and just had
  # containerd run "runc" in that "WorkDir" we saw earlier, the "Image" is blank:
  "Image": "",
  # This is how Docker told containerd to run the container using runc:
  "Runtime": {
      "Name": "io.containerd.runc.v2", ... },
  # This is the Open Container Initiative (OCI) Spec for the container:
  "Spec": {
    # containerd can handle multiple OCI spec revisions transparently for us!
    "ociVersion": "1.0.2-dev",
    "process": {
      "terminal": true,  # This is the "-t" argument we passed to "docker run".
      "user": { "uid": 0, other user ID settings ... },
      # Here's our "bash" command we used in "docker run":
      "args": ["bash"],
      # Here's all the environment variables for our process:
      "env": ["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin", ...],
  # "root" is where the root directory of the container lives on the host:
  "root": {"path": "/var/lib/docker/overlay2/${UGLY_ID_AGAIN}/merged"},
  # Here's where all the declaration for bind mounts in the container are:
  "mounts": [ {"destination": "/proc", ...}, {"destination": "/dev"} ]

# Despite the container ecosystem getting very confusing at times, remember:
# CONTAINERS ARE NOT MAGIC, JUST PROCESSES WITH ADDED STEPS!
# Every concept that goes into making a container is easily within your grasp.
```

Listing 16: Poking around Docker Containers. (3/3)

5.2 Building Container images with Dockerfiles.

In this section, we will be looking at how to build a Docker image for a simple Python Flask app:

```
#!/usr/bin/env python3

"""
Simple Flask app which lists/cats directories/files on the host.

Prerequisites on Ubuntu 22.04:
    - apt install -y python3 python3-pip
    - pip3 install Flask

Args and examples:
    - python3 /path/to/the/flask_app.py LISTEN_ON_HOST LISTEN_ON_PORT
    - python3 flask_app.py localhost 8080
    - python3 flask_app.py 10.7.2.3 9000
    - python3 flask_app.py 0.0.0.0 9192
"""

import json, os, sys
import flask

# Declare Flask app:
app = flask.Flask(__name__)

@app.route('/', defaults={'path': ''})
@app.route('/<path:path>')
def get_dirpath(path):
    path = f"/{path}"
    if os.path.isdir(path):
        # List entries within directories:
        return json.dumps(sorted(os.listdir(path)), indent=4)
    elif os.path.isfile(path):
        # Return contents of files:
        with open(path, "r") as fin:
            return fin.read()
    else:
        # 404 if the path is a link or does not exist:
        flask.abort(404)

if __name__ == "__main__":
    print(f"sys.argv were: {sys.argv}")
    if len(sys.argv) < 3:
        raise ValueError(
            f"USAGE: python3 {sys.argv[0]} LISTEN_ON_HOST LISTEN_ON_PORT")
    hostname = sys.argv[1]
    port = int(sys.argv[2])

    app.run(hostname, port)
```

Listing 17: Simple Python Flask App we'll be containerizing.

5.2.1 Running a Flask app manually in a Docker container.

```
# After saving our Flask app to a file called "flask_app.py", let's take it for a spin:
$ sudo apt install -y python3 python3-pip curl
$ sudo pip3 install Flask
$ python3 flask_app.py localhost 8080

# In a separate shell on the Host, we can use "curl" to make HTTP calls to our app:
$ curl http://localhost:8080/
[
    "bin",
    "boot",
    "dev",
    ...
]

# Let's also try "cat"-ing a file:
$ curl http://localhost:8080/etc/hostname
<your hostname>

# Naturally, there's nothing stopping us from repeating the steps within a Docker container:
$ docker run -ti --name flask-container --hostname flask-container ubuntu:22.04 bash
[C] $ apt update && apt install -y python3 python3-pip iproute2 curl
[C] $ pip3 install Flask

# Now, in order to copy over the flask app from the Host, we can use "docker cp":
$ docker cp /path/to/host/flask_app.py flask-container:/flask_app.py

# We can now run the Flask app in the container:
[C] $ python3 /flask_app.py localhost 8080
<app should start successfully>

# Note that because we started the app with "localhost", we cannot access it from the Host,
# as Flask will refuse any connections unless they come from within the container itself:
$ docker exec -ti flask-container curl http://localhost:8080/etc/hostname
flask-container

# It's finally running, but those were a lot of steps and work to have to do every
# time a new version of Ubuntu or our "flask_app.py" file would get released.
# What would be nice if there was a way to...codify these steps into a file for Docker?
```

Listing 18: Manually running the Flask app in a Docker container.

5.2.2 Dockerfiles: just app installation steps executing in containers.

```
# As we've seen, installing dependencies and running our Flask app works basically
# identically in a container as it does on the Host, just with extra steps...
# It would really suck to have to do it every time we want to deploy our app through!

# Fortunately, we can codify the exact installation steps into a so-called "Dockerfile".
# A Dockerfiles is a list of instructions (called "STANZAS") Docker will perform
# within a container for you, and offer you a "Container Image" to re-use at will.

# Let's write a trivial Dockerfile for our Flask app:
$ mkdir build; cd build; cp /path/to/flask_app.py ./

# Please create a file named "Dockerfile" with the following contents:
$ cat Dockerfile
# This will tell Docker to start from the ubuntu:22.04 image as a base:
FROM ubuntu:22.04

# This will copy our flask app file into the container similar to "docker cp":
# NOTE: you will need to make sure that "./flask_app.py" is in the dir you build!
COPY ./flask_app.py /flask_app.py

# Now to add the commands needed to install our dependencies:
RUN apt update && apt-get install -y python3 python3-pip iproute2 curl
RUN pip3 install Flask

# This will tell Docker what command to run as the "entrypoint" of the container.
# When doing "docker run/start/restart", Docker will call this exact command:
ENTRYPOINT python3 "/flask_app.py" localhost 8080

# Save the above in a file named "Dockerfile" and let's try building an image!
# The "docker build" command will automatically search for files named "Dockefile"
# in the current working directory, and send them to the "dockerd" service to be built.
# We will need to come up with a name for the image, of the form "$NAME:$TAG".
# The "$TAG" part usually represents a so-called "SemVer version string". (e.g. 1.0.0)
$ docker build -t flask-app:1.0.0 .
<Docker will run each install step in a new container per "image layer">

# If you're quick enough, you may be able to see the temporary containers doing the build:
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS
# NOTE: this one was running the "RUN apt update" stage of the Dockerfile:
db6458621f2e   d9ec2db501a9   "/bin/sh -c 'apt upd..."   9 seconds ago   Up 8 seconds

# Once Docker's done building the image, we should be able to run it:
$ docker run --detach --name flask-container --hostname flask-container flask-app:1.0.0

# Check the app works from within the container:
$ docker exec -ti flask-container curl http://localhost:8080/etc/hostname
flask-container
```

Listing 19: Dockerfiles intro.

5.2.3 Defining Build ARGs and ENV variables in Images.

```
# One sad thing you may have noticed is that we've been running the "curl" commands
# to test our app within the Docker container itself. This is due to the fact that
# our image ENTRYPOINT tells Flask to only listen for connections on "localhost".
# Let's get the IP address address of the container we created in the previous section:
$ docker exec flask-container ip a  # Get IP of running container.
$ curl http://172.17.0.2:8080/etc/hostname
# Note that it explicitly said "connection refused", and not "no route to host" etc...
curl: (7) Failed to connect to 172.17.0.2 port 8080 after 0 ms: Connection refused

# We need to tell the Flask app to listen for all incoming connections on "0.0.0.0".
# We can do this by declaring a build argument with "ARG". Simply add this to the Dockerfile:
$ cat Dockerfile
<same Dockerfile steps as before, but we will be changing the ENTRYPOINT>

# This added line will define a build argument named "LISTEN_ON_HOST" which we can
# control during the "docker build" stage using "--build-arg LISTEN_ON_HOST=0.0.0.0".
ARG LISTEN_ON_HOST="localhost"

# This line will make "LISTEN_ON_HOST" available as an environment variable in the
# "ENTRYPOINT", so we will be able to pass "$LISTEN_ON_HOST" to "flask_app.py".
ENV LISTEN_ON_HOST=${LISTEN_ON_HOST}
ENTRYPOINT python3 "/flask_app.py" $LISTEN_ON_HOST 8080

# Now let's build two versions of the image: a localhost-only dev build, and a full one:
$ docker build --build-arg LISTEN_ON_HOST=0.0.0.0 -t flask-app-anyhost:1.0.0 .
$ docker build --build-arg LISTEN_ON_HOST=localhost -t flask-app-localhost:1.0.0 .

# Now let's run both and see the differences:
$ docker run -d --name flask-localhost --hostname flask-localhost flask-app-localhost:1.0.0
$ docker run -d --name flask-anyhost --hostname flask-anyhost flask-app-anyhost:1.0.0

# Get the IP addresses of both with "docker exec flask-localhost ip a" and try:
$ curl http://172.17.0.3:8080/  # IP of flask-localhost container.
curl: (7) Failed to connect to 172.17.0.3 port 8080 after 0 ms: Connection refused
$ curl http://172.17.0.4:8080/  # IP of flask-anyhost container.
<works>

# As mentioned, LISTEN_ON_HOST is nothing more than an environment variable:
$ docker exec flask-anyhost env | grep LISTEN_ON_HOST

# We can even pass a random LISTEN_ON_HOST during "docker run" using "--env":
$ docker run -d --name flask-randomhost --hostname flask-randomhost \
    --env LISTEN_ON_HOST="flask-randomhost" flask-app-anyhost:1.0.0

# This will refuse connections on anything except "flask-randomhost":
$ docker exec flask-randomhost curl http://localhost:8080
curl: (7) Failed to connect to localhost port 8080 after 1 ms: Connection refused
$ docker exec flask-randomhost curl http://flask-randomhost:8080
<works>
```

Listing 20: Dockerfile build ARGs and ENV variables.

5.3 Docker Image Dissection.

We've already been introduced to the OCI image spec in Section 4.4.1, but let's actually dive deeper into its inner workings:

```
# As mentioned, images have layers, each representing a "stage" in the Dockerfile:
$ docker history flask-app-anyhost:1.0.0
IMAGE          CREATED       CREATED BY                                      SIZE
# NOTE: Some layers are just "configuration layers" with zero actual file data:
545bbacaf60c   8 hours ago   /bin/sh -c #(nop)  ENTRYPOINT ["/bin/sh" "-c... 0B
f0a5166fb92d   8 hours ago   /bin/sh -c #(nop)  ENV LISTEN_ON_HOST=0.0.0.0   0B
10fb988e6c8b   8 hours ago   /bin/sh -c #(nop)  ARG LISTEN_ON_HOST=localh... 0B
# NOTE: here's the layers where we installed the Python and Flask dependencies:
4034b102502c   9 hours ago   /bin/sh -c pip3 install Flask                   4.67MB
e9b966c06beb   9 hours ago   /bin/sh -c apt update && apt-get install -y ... 401MB
# NOTE: and here is where our Flask app file is COPY'd!
d9ec2db501a9   9 hours ago   /bin/sh -c #(nop) COPY file:15902739037d0aea... 900B
# This layer was inherited from the "ubuntu:22.04" image we used as a base:
3b418d7b466a   6 weeks ago   /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<some more "base layers" added automatically by Docker which we skip here>

# We can "export" any image using "docker save". This will contain all the layers:
$ docker save flask-app-anyhost:1.0.0 -o flask-app.oci

# The OCI image is just a TAR archive we can freely unpack:
$ mkdir /tmp/unpacked-flask-app && tar -xvf flask-app.oci -C /tmp/unpacked-flask-app/

# You'll note there's a "manifest.json" file which points to a file named after the ID:
$ cat /tmp/unpacked-flask-app/manifest.json
[{"Config":"${YOUR_IMAGE_ID}.json" ...

# The JSON named after the image ID lists everything from CPU architecture,
# User settings, the Entrypoint for the image, and more, check it out:
$ cat /tmp/unpacked-flask-app/${YOUR_IMAGE_ID}.json

# You will also note that there are only 4 actual layers. One is the base Ubuntu
# layer, and the other 3 layers are the Dockerfile commands which actually
# changed data. (i.e. the COPY and two RUNs, since ARG or ENV do not add files)
# Unfortunately the names of the layers are not correlated, but we can bet the
# one with the "flask_app.py" file has to be the smallest, so we can do this:
$ du -s /tmp/unpacked-flask-app/*
# Please pick the layer directory with the smallest size and cd into it.

# The "VERSION" file defines the OCI image format version:
$ cat VERSION
# The "json" file defines the same Entrypoint/User settings etc as IMAGE_ID.json:
$ cat JSON
# And finally, the "layer.tar" file contains the actual files of the layer:
$ tar -xvf layer.tar
flask_app.py
$ cat flask_app.py
<the original contents of our flask_app.py file>
```

Listing 21: OCI Container Image Dissection.

5.3.1 Image Layer Optimisation and Multi-Stage Builds.

As we've seen, each line of a Dockerfile represents a command (or "stanza") which somehow affects the image. This can be superficial setting changes (e.g. `WORKDIR`, which just sets the working directory for all following stanzas), or they can be commands which actually affect image files (e.g. "`RUN apt install ...`").

Naturally, there has emerged a set of "best practices" when writing Dockerfiles:

- **layer ordering and caching**: presuming our Dockerfile has a handful of separate "`RUN ...`" stages which add/modify files, we should try to put more stable layers at the bottom, and layers changing more often on top. This will allow Docker to re-use image layers more often. (e.g. if our `flask_app.py` changes often, it should be one of the last layers we add)
- **layer merging**: when running multiple commands which change actual files (e.g. "`apt update; apt upgrade`"), it may be worth considering unifying multiple layers into one by simply going "`cmd1 && cmd2`", or even writing a script which we can execute in a single RUN command. This will prevent the final image from having multiple layers which could be just one layer, thus saving on storage space and image pull times.
- **multistage builds**: sometimes ywas as informative as it was lighthearted, and itou need to `RUN` some steps on one base layer (e.g. you need all the tools in `ubuntu:22.04` to build an app), but the app can run on a more minimal base image (e.g. `alpine:latest`). Multistage builds allows you to declare separate build stages to be done in one container, and the final image being based on another.

Here's how an optmizied multistage build for our Flask app from Listing 17 may look like:

```
# NOTE: we can name any build stage using "as <Name>" after "FROM":
FROM ubuntu:22.04 as Stage1

# NOTE: we removed "WORKDIR /" since it was redundant.

# Note we squished the "pip3" in the same RUN command as the "apt"s:
RUN apt update \
    && apt-get install -y python3 python3-pip iproute2 curl \
    && pip3 install Flask

# NOTE: we moved the flask_app.py file COPY lower down, since there's no point
# in re-installing all the dependencies with "apt" every time we add a "print()".
COPY ./flask_app.py /flask_app.py

ARG LISTEN_ON_HOST="localhost"
ENV LISTEN_ON_HOST=${LISTEN_ON_HOST}

# NOTE: "scratch" is basically a completely blank base image.
# The resulting image we build will be based on scratch, not Ubuntu.
FROM scratch

# This will literally just copy everything from Stage1 into our empty scratch space,
# which might seem redundant, but will result in an image with a single data layer.
# (The other data-less "ARG" or "ENV" layers will still exist, just not take space)
COPY --from=Stage1 / /

ENTRYPOINT python3 "/flask_app.py" ${LISTEN_ON_HOST} 8080

# Try to "docker build -t flask-app-optimized:1.0.0 ." and "docker history" this image!
```

Listing 22: Flask App Multistage Dockerfile.

5.4 Docker Networking modes.

Docker offers 4 Networking modes for its users to start containers in:

- **none**: the base "no networking mode" creates the container in a dedicated network namespace with nothing but a loopback interface, so it cannot talk to anything over network except for itself.
- **host**: "Host networking mode" basically skips network namespace creation for the container and the processes simply share the Host's network namespace. This means that all the processes in the container are directly exposed to the Internet, as well as see all traffic on the Host. (including any traffic to/from the Host and other containers)
- **container**: "Container networking mode" simply starts the container in the same network namespace as another existing container. This allows processes in both containers to freely speak to each-other over the network, but added configuration is needed for them to reach the Host/general Internet.
- **bridge**: "Bridge(d) networking mode" still starts the containerized processes in a dedicated network namespace, but also automatically creates a pair of bridged virtual network interfaces in both the Host and Container network namespaces and sets up IPv4/v6 forwarding between the Host and Container interfaces so that the Container may access the Internet. (exactly as seen in Listing 8)

Let's quickly go through the basic networking modes first:

```
# First off, we can list available networks on our Docker Host using:
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
c907b70ea834   bridge    bridge    local
78907a706629   host      host      local
6c31762f1478   none      null      local

# The "none" network is self-explanatory: a new network namespaces with NOTHING:
$ docker run -d --name flask-forever-alone --net=none flask-app-localhost:1.0.0
$ docker exec flask-forever-alone ip a
<only the "lo" loopback interface :( >
# This is because the container is in its own empty network namespace:
$ docker exec flask-forever-alone lsns -t net
        NS TYPE NPROCS PID USER    NETNSID NSFS COMMAND
4026532456 net       3   1 root unassigned      /bin/sh -c python3 "/flask_app.py" ...

# Pretty useless right? Pretty much as secure as Docker containers are gonna get tho!
# The opposite of this would be "--net=host", where any processes are launched in
# *the same network namespace* as the Host, granting it full rights to do whatever:
$ docker run -d --name flask-forever-onhost --net=host flask-app-anyhost:1.0.0
$ docker exec flask-forever-onhost lsns -t net
        NS TYPE NPROCS PID USER    NETNSID NSFS COMMAND
# NOTE: running "lsns -t net" on the Host should show this to be the same net Namespace.
4026531840 net       3   1 root unassigned      /bin/sh -c python3 "/flask_app.py" ...

# Because the container is sharing the host network namespace, any incoming network
# connection on a port on the host could potentially go to a listening container process.
$ curl http://${HOST_IP}:8080/
<will list the files from the Container, since the fs/mounts are still namespaced>
```

Listing 23: Docker Basic Networking Modes.

```
# Host networking mode is cool and all, but the whole purpose or running things
# in containers is for added security, so "--net=host" just won't cut it...
# Bridge networking mode offers a way to run containers in an internal network
# isolated from the Host's network namespace, while still setting up virtual
# interfaces and forwarding rules so the containers can reach the Host/Internet.

# Let's inspect the default bridged Docker network named, you guessed it, "bridge":
$ docker network ls -f driver=bridge
NETWORK ID     NAME      DRIVER    SCOPE
c907b70ea834   bridge    bridge    local
$ docker network inspect bridge
[{
    ...
    "Name": "bridge",
    "Driver": "bridge",
    "IPAM": {
        "Driver": "default",
        # NOTE: IP range for the network is defined here:
        "Config": [{ "Subnet": "172.17.0.0/16", "Gateway": "172.17.0.1" }]
    },
    ...
     "Options": {
        # NOTE: enables Network Address Translation (NAT) for the network:
        "com.docker.network.bridge.enable_ip_masquerade": "true",
        "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
        # NOTE: this is the name of the L2 bridge container NICs will be connected to:
        "com.docker.network.bridge.name": "docker0",
        ...

# Notice the "docker0"? It's the name of the Host L2 bridge containers will connect to:
$ ip a show docker0
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    # NOTE: this is the IPv4 address containers can reach the Host on:
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0

# Let's run a quick test container:
$ docker run -d --name flask-bridge-test --net=bridge flask-app-anyhost:1.0.0
$ docker exec flask-bridge-test ip a
100: eth0@if101: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0

# The Host (172.17.0.1) should be able to access the container (172.17.0.3):
$ curl http://172.17.0.3:8080/  # should work.

# A service only accessible in the Docker host is not that useful, we'd ideally
# like to be able to "expose" a container port on the Host to be widely-accessible:
$ docker rm -f flask-bridge-test  # delete old container
# "-p 80:8080" will automatically foward IPv4 traffic from Host port 80 to
# the container's 8080, making the Flask app accessible from the Host.
$ docker run -d --name flask-bridge-test --net=bridge -p 80:8080 flask-app-anyhost:1.0.0
$ curl http://${HOST_IP}:80
<should list the "/" directory of the container>
```

Listing 24: Docker Bridge Networking Showcase. (1/2)

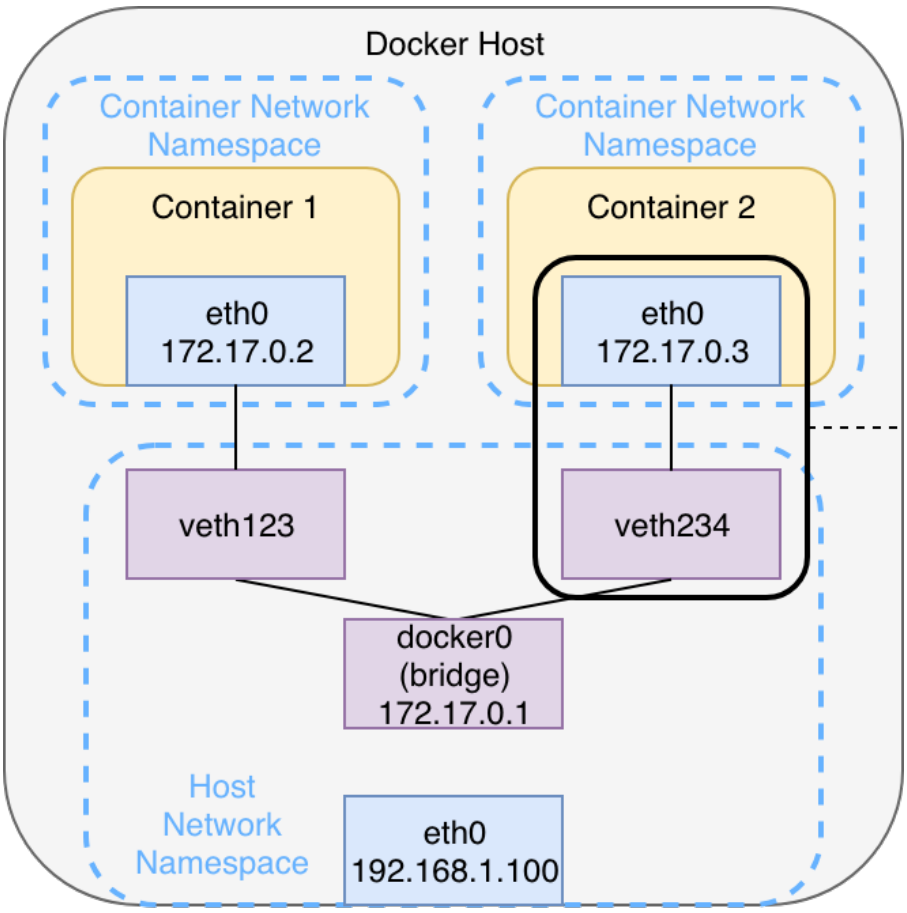

Figure 5.2: Docker Bridged network diagram.

```
# This begs the question: how are the ports being forwarded? Remember iptables?
# Docker uses iptables to create a Network Address Translation (NAT) for the containers:
$ iptables -t nat -S
# NOTE: Docker defines a new rule chain named "DOCKER":
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
# This rule sets up the sNAT on the Docker subnet range:
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
...

$ iptables -S DOCKER
# These rules set up the DNAT, ensuring it passes through the "DOCKER" chain.
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
...

# Note that the Docker subnet is arbitrary, and we could use any private range:
$ cat /var/lib/docker/network/files/local-kv.db | strings | grep 172.17.0.0/16
<this is a binary db file Docker uses, but the output is barely legible>
```

Listing 25: Docker Bridge Networking Showcase. (2/2)

5.5 Docker Volumes and Mounts.

```
# Considering containers are so volatile, how can we better ensure their data?
# Docker allows for the creation of "persistent volumes" which can be freely
# mounted and used by containers, but can be managed individually too:
$ docker volume create myvolume
$ docker volume inspect myvolume
[{
  "Name": "myvolume",
  "Driver": "local",
  "Mountpoint": "/var/lib/docker/volumes/myvolume/_data",
}]

# Wait a minute, is it really nothing but a directory on the local system?
$ file /var/lib/docker/volumes/myvolume/_data
/var/lib/docker/volumes/myvolume/_data: directory

# Let's try to run a container with it using '-v ${VOLUME_NAME}:${PATH_IN_CONTAINER}':
$ docker run -ti -v myvolume:/myvolume-in-container --name volumetest ubuntu:22.04 bash
[C] $ echo "This file was saved from the container!" > /myvolume-in-container/file.txt

# Checking on the host, we can see the the file directly:
$ cat /var/lib/docker/volumes/myvolume/_data/file.txt
This file was saved from the container!

# All that Docker did was just "mount --bind" that directory in the container's
# root filesystem before starting the containerized process's mount namespace:
[C] $ mount | grep myvolume
/dev/mapper/ubuntu--vg-ubuntu--lv on /myvolume-in-container type ext4 (rw,relatime)

# Naturally, there's nothing stopping Docker from treating any directory as a volume.
# Allowing the container to modify files in a host directory, however, defeats most
# of the purpose of container, so we can also add ":ro" to make it Read-Only:
$ docker run -ti -v /etc:/hostetc:ro --name volumetest2 ubuntu:22.04 bash
[C] $ cat /hostetc/hostname
<hostname of your Docker host>
[C] $ echo "trololol.com" > /hostetc/hostname
bash: /hostetc/hostname: Read-only file system

# Inspecting the two containers, we can see the JSON fields responsible clear as day:
$ docker inspect volumetest{,2} | grep Mounts -A 5
# Mounts[0] for "-v myvolume:/myvolume-in-container":
  "Type": "volume",
  "Name": "myvolume",
  "Source": "/var/lib/docker/volumes/myvolume/_data",
  "Destination": "/myvolume-in-container",
# Mounts[0] for "-v /etc:/hostetc:ro":
  "Type": "bind",
  "Source": "/etc",
  "Destination": "/hostetc",
  "Mode": "ro",
```

Listing 26: Docker Volumes Showcase.

5.6 Docker Registry: DockerHub vs. Local Registry Setup.

A Container Registry is nothing more than a server which stores and shares Container image layers.

By default, Docker will pull images from DockerHub, which is a mostly open Container Registry managed by the Docker company, and houses most Container images you'd ever want, including the Ubuntu Image we've been using.

Unfortunately, the standard Docker Engine does not come with a dedicated Registry service, and setting one up is non-trivial, so we will be relying on the script in Listing 27.

```
#!/bin/bash

set -eux

if [ $UID -ne 0 ]; then echo "Must be run as root!"; exit 1; fi

apt-get install -y apache2-utils

# Host directory to create the certs/auth in and mount to the registry container.
# Feel free to give the script different values for these vars if you like.
CERTS_DIR="${CERTS_DIR:=/etc/docker/testregcerts}"
REGISTRY_USER="${REGISTRY_USER:=testuser}"
REGISTRY_PASSWORD="${REGISTRY_PASSWORD:=Passw0rd}"

# Create the self-signed x509 certificates we'll be using:
mkdir -p "$CERTS_DIR"
openssl req \
  -newkey rsa:4096 -nodes -sha256 -keyout "${CERTS_DIR}/domain.key" \
  -addext "subjectAltName = DNS:myregistry.domain.com" \
  -x509 -days 365 -out "${CERTS_DIR}/domain.crt"

# This will encrypt the password so it won't be saved as plaintext by Docker.
AUTH_DIR="${CERTS_DIR}/auth"
mkdir -p "$AUTH_DIR"
htpasswd -bnB "$REGISTRY_USER" "$REGISTRY_PASSWORD" > "${AUTH_DIR}/htpasswd"

# Run registry container with mounts/env vars required for auth.
# The `-v source:target` arguments mount the host directories in the containers.
# The `-e VAR=val` arguments pass the necessary env variables for the Registry.
# Note that the `-e VAR=/path` arguments refer to paths WITHIN the registry container.
docker container run -d \
        -p 5000:5000 --name registry_basic \
        -v "${AUTH_DIR}":"${AUTH_DIR}" -v "${CERTS_DIR}":"${CERTS_DIR}" \
        -e REGISTRY_AUTH=htpasswd \
        -e REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm" \
        -e REGISTRY_AUTH_HTPASSWD_PATH="${AUTH_DIR}/htpasswd" \
        -e REGISTRY_HTTP_TLS_CERTIFICATE="${CERTS_DIR}/domain.crt" \
        -e REGISTRY_HTTP_TLS_KEY="${CERTS_DIR}/domain.key" \
        registry:latest

# Cat the creds for the user:
echo "Your 'user / pass' is:  ${REGISTRY_USER} / ${REGISTRY_PASSWORD}"
```

Listing 27: Docker Registry Setup Script.

```
# before anything, not that the default Docker installation will pull from DockerHub:
$ sudo docker info | grep Registry
 Registry: https://index.docker.io/v1/

# While DockerHub is nice, let's try to deploy our own Registry to be safe!
# Fortunately, there is an official image named "registry" on DockerHub.
$ docker pull registry:latest

# Unfortunately, user/pass auth is hard to set up, so we'll be using our script:
$ sudo bash /path/to/setup_registry_with_auth.sh
<the script will tell you the user/pass once it finishes>

# The script will run the "registry:latest" image and expose port 5000.

# To log into our registry, simply run this with the creds provided by the script:
$ docker login localhost:5000
<user/pass should be testuser/Passw0rd by default>

# There are currently no images on our registry, so let's push our Flask image.
# To do so, we must "tag" the image to indicate it lives on our "localhost" registry:
$ docker image tag flask-app:1.0.0 localhost:5000/flask-app:1.0.0

# Note that multiple tags can point to the same image no problem:
$ docker image ls
docker image ls
REPOSITORY                  TAG       IMAGE ID       CREATED        SIZE
flask-app                   1.0.0     378fd1f4eb45   11 hours ago   483MB
localhost:5000/flask-app    1.0.0     378fd1f4eb45   11 hours ago   483MB

# Pushing the tag without the repository prefix will try to upload to DockerHub:
$ docker push flask-app:1.0.0
The push refers to repository [docker.io/library/flask-app]
# ...we'll likely get an access denied, since we're not logged into DockerHub:
denied: requested access to the resource is denied

# The key is giving it the tag which specifically mentions our local Registry:
$ docker push localhost:5000/flask-app:1.0.0
<should see nice progress bars for each image layer being uploaded>

# Let's try deleting the image on the Docker Engine and re-pulling it:
$ docker image rm flask-app:1.0.0
$ docker run -ti flask-app:1.0.0
Unable to find image 'flask-app:1.0.0' locally
<will try searching for the image on DockerHub>

# Again, we need to tell Docker that the image is from the local registry:
$ docker run -ti localhost:5000/flask-app:1.0.0
<missing image layers will be downloaded and the app should start successfully>

# We can now freely distribute images between multiple Docker hosts with our Registry!
```

Listing 28: Docker Registry Local Setup.

5.7 Docker Compose: like scripts, but for `docker`, and in YAML...

As we've seen, containers have an awful lot of options to running them, and `docker run` commands can grow to become very unwieldy.

`docker-compose` is a Docker tool which allows us to define all the options for multiple containers in a pleasant-to-write YAML file known as a Compose File, let's see how that looks:

```
# Recall the horrendous Docker command we used to deploy the test registry?
$ export CERTS_DIR="/etc/docker/testregcerts"
$ export AUTH_DIR="${CERTS_DIR}/auth"
$ docker container run -d -p 5000:5000 --name registry_basic \
        -v "${AUTH_DIR}":"${AUTH_DIR}" -v "${CERTS_DIR}":"${CERTS_DIR}" \
        -e REGISTRY_AUTH=htpasswd \
        -e REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm" \
        -e REGISTRY_AUTH_HTPASSWD_PATH="${AUTH_DIR}/htpasswd" \
        -e REGISTRY_HTTP_TLS_CERTIFICATE="${CERTS_DIR}/domain.crt" \
        -e REGISTRY_HTTP_TLS_KEY="${CERTS_DIR}/domain.key" registry:latest

# It objectively hurts the eyes, but we can encode all that info into a compose
# file using YAML. NOTE: use 2 SPACES for every indent level. NO '\t'!
$ cat registry-compose.yaml
version: '3'  # Note: the "version" of the compose file is important!
services:     # we can declare any number of containers under "services".
  registry-basic-auth:  # This will be used as the name for the container.
    # It's the same "registry:latest" image notation as before:
    image: registry:latest
    ports:
      - "5000:5000"  # Here's the "-p 5000:5000" port mapping.
    volumes:
      # These are the bind mounts declared via "-v host_dir:container_dir:ro"
      - /etc/docker/testregcerts:/regcerts:ro
      - /etc/docker/testregcerts/auth:/auth:ro
    environment:
      # These are all the env vars:
      - REGISTRY_AUTH=htpasswd
      - REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm"
      - REGISTRY_HTTP_TLS_KEY="/regcerts/domain.key"
      - REGISTRY_AUTH_HTPASSWD_PATH="/auth/htpasswd"
      - REGISTRY_HTTP_TLS_CERTIFICATE="/regcerts/domain.crt"


# After installing docker-compose, we can use "docker-compose up" to deploy:
$ sudo apt install docker-compose
$ docker-compose -f registry-compose.yaml up --detach
Creating root_registry-basic-auth_1 ... done

# Note that "docker-compose" relies on the compose file to "remember" the topology
# of your services, so you will need to pass the compose file every time:
$ docker-compose -f ./registry-compose.yaml ps
           Name                         Command               State          Ports
-------------------------------------------------------------------------------------------
root_registry-basic-auth_1   /entrypoint.sh /etc/docker ...   Up      0.0.0.0:5000->5000/tcp
```

Listing 29: Docker Compose Showcase.

Chapter 6

Kubernetes and Container Orchestration Teaser.

We've seen how Container Engines (4.2) offer end-to-end solutions for running multiple Containers on a single server, so the next logical level is defining a **Container Orchestrator** to...well...orchestrate the deployment of containers on multiple Container Engine hosts!

While **Kubernetes and other Container Orchestrators will be covered more in-depth in future Labs**, let's give a quick overview of the features we'd expect from any Container Orchestrator.

- **dynamically add/remove nodes**: any Container Orchestrator should be able to dynamically add "nodes" (aka Container Engine servers) and dynamically "evacuate" any containers running on nodes when needed.
- **integrated load balancing**: given they manage multiple Engine hosts, Orchestrators should provide the ability to scale our applications up (i.e. deploy multiple containers of the same image on multiple nodes) and also load-balance traffic to each container.
- **monitoring**: ability to set alerts on container events, or define recovery procedures which are triggered automatically in case the user applications have issues.
- **autoscaling**: depending on how much traffic a service hosted on our Orchestrator is seeing, it could automatically scale the service up/down by adding/removing containers for the service.

6.1 Docker Swarm.

In 2014, Docker Swarm was one of the first and most obvious implementations of a Container Orchestrator: it's simply a service which manages consensus between multiple Docker Engine hosts.

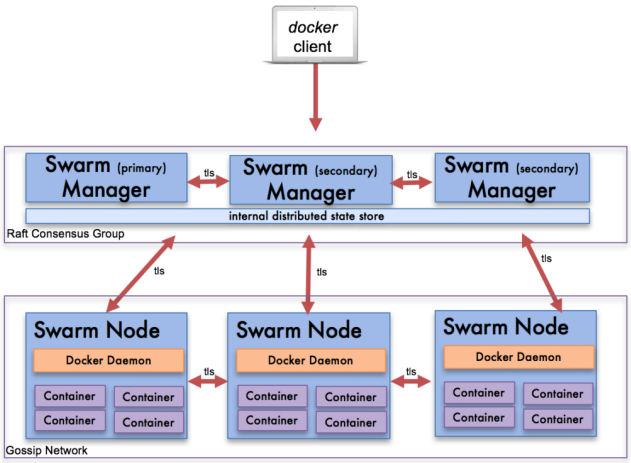

Figure 6.1: Docker Swarm Architecture.

6.2 Apache Mesos.

Mesos is a much more abstract and research-driven project which aims to manage compute clusters for basically any type of task, with the support of "Docker container tasks" making it a very viable yet quite convoluted way to schedule containers on Docker hosts.

While looking at the architecture in Image 6.2, try to mentally replace "executor" with "Docker Engine" and "task" with "container".

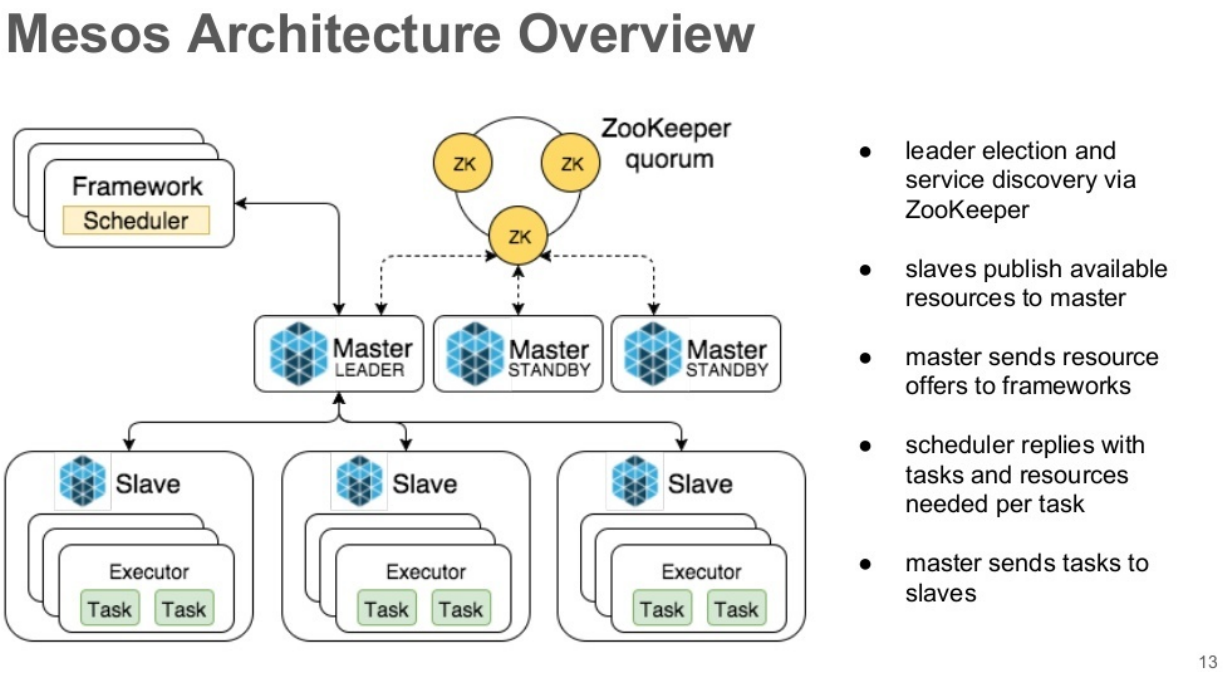

Figure 6.2: Mesos high-level architecture.

6.3 Kubernetes: smooth sailing for Container Orchestration.

History has shown the winner of the Container Orchestrator race to be Kubernetes, which allows users to run, monitor, scale, upgrade, and give access to OCI containers on multiple hosts, called "Kubernetes Nodes". It achieves this through a layered arhitecture with multiple components:

- **Kubernetes clients** like the `kubelet` command are used to send YAML configurations of the services we want it to run to the Control Plane, which will later deploy them on the Data Plane.
- the **Control Plane** (aka brains of the operation) is composed of:
  - the **API server**: the endpoint "kubectl" connects to. Any user calls will lead to changes in the cluster state in **etcd**.
  - **etcd**: a distributed key-value store (i.e. really fast clustered database with a consensus algorithm) to ensure system settings are propagated across the whole Control Plane reliably and safely.
  - **Controller Manager**: a component which controls the lifecycle of internal resources defined in the configurations in **etcd**, as well as user-defined resources knows as Custom Resource Definitions. (CRDs)
  - **Scheduler**: decides on which Node from the Data Plane to schedule the work that the Controller Manager decided needed doing.
- **Data Plane** refers to the the layer where user/application data is stored/moved. (aka the Kubernetes Nodes actually running our containers) It is comprised of Servers (or "Nodes") which are all running the following:
  - **kubelet**: the service installed on every Node which actually calls the Container Runtime to run containers on the server.
  - **kube-proxy**: manages networking across the nodes to provide features like port forwarding for the containers, load balancing, etc...

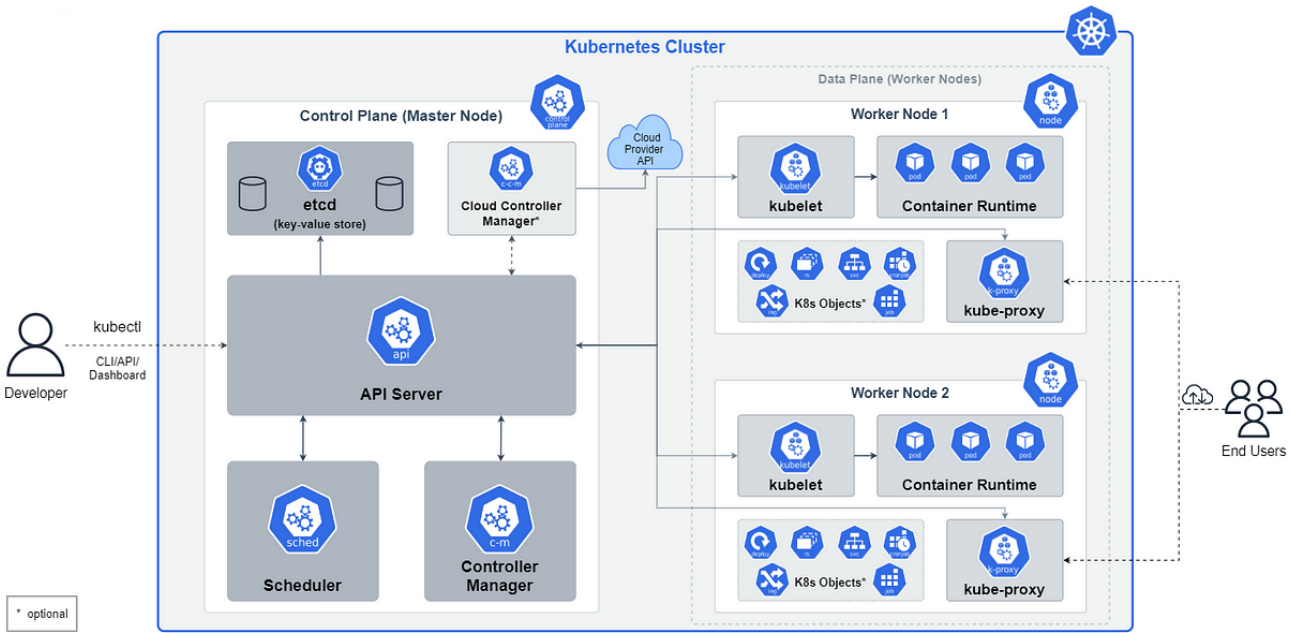

Figure 6.3: Kubernetes Architetural Overview.

6.4 Rancher.

Rancher is basically just a vendorized Kubernetes with a nice UI, but it is worth mentioning since they are currently one of the industry leaders when it comes to application deployment platforms.

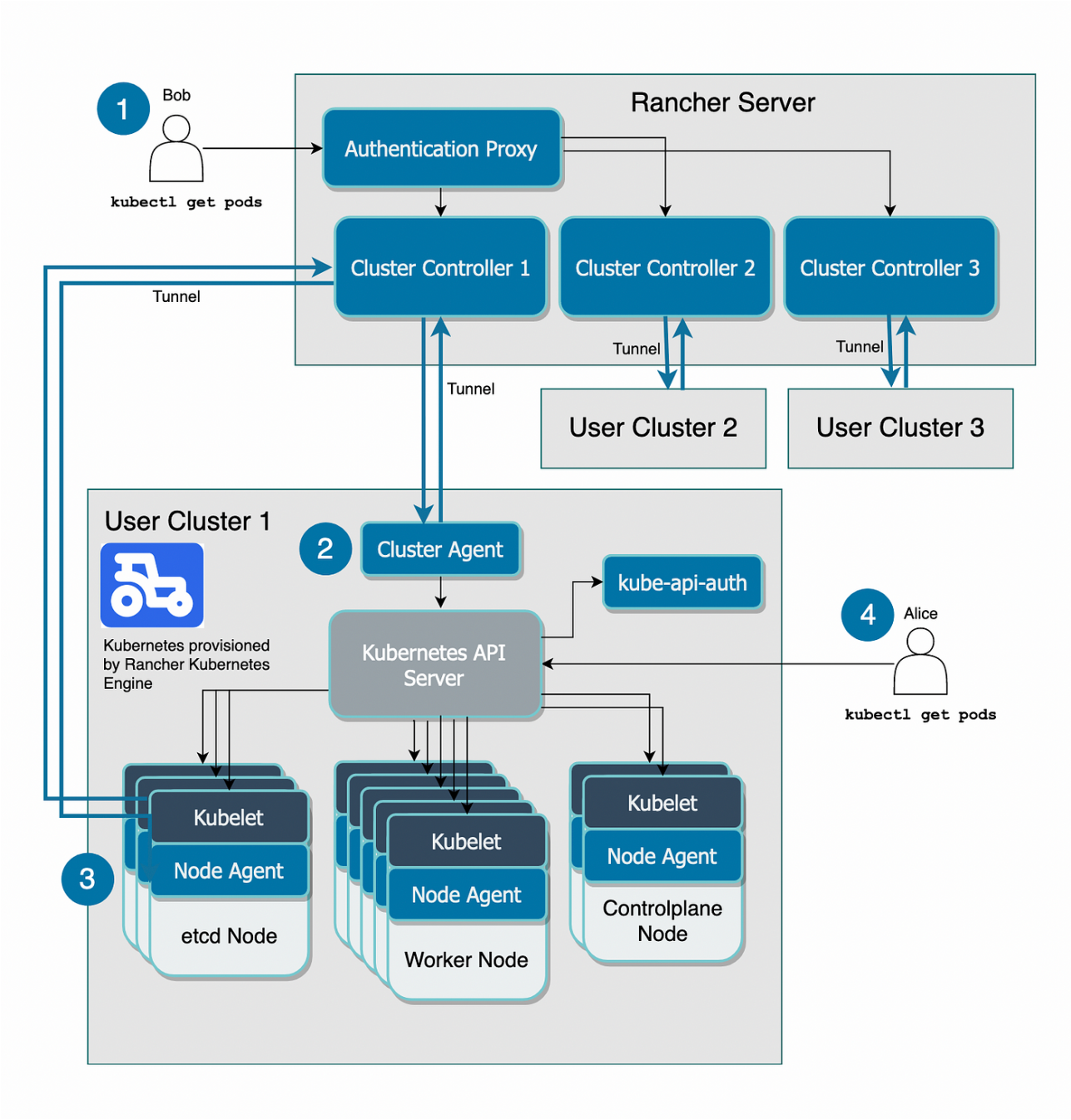

Figure 6.4: Racher architecture.

Chapter 7

Exercises for the Reader

Thanks again for making it this far, we hope this document sparked a deeper understanting and appreciation for the Container Ecosystem so that you may continue your journey through containerization on your own.

This chapter will list some exercises which will greatly help in deepening some of the concepts covered, and it is recommended that you at least give them a shot. Who knows, maybe you're a bit of a Container Orchestrator yourself!

Remember: trying and failing miserably is the very first step towards succeeding. Questions are always welcome.

7.1 Improving our Image Building skills.

As we've seen in Listing 19, container images are built using so-called "Dockerfiles", where each command (or "stanza") used in the Dockerfile represents a layer of the image.

Here's the Dockerfile for the Flask app (17) we've written:

```
FROM ubuntu:22.04

WORKDIR /
COPY ./flask_app.py /flask_app.py

RUN apt update && apt-get install -y python3 python3-pip iproute2 curl
RUN pip3 install Flask

ARG LISTEN_ON_HOST="localhost"
ENV LISTEN_ON_HOST=${LISTEN_ON_HOST}

ENTRYPOINT python3 "/flask_app.py" ${LISTEN_ON_HOST} 8080
```

Listing 30: Flask App Baseline Dockerfile.

There are a number of improvemets you could bring to this image:

- make the port number a build ARG/env VAR nameed "LISTEN_ON_PORT".
- change the image base ("FROM") to something smaller like Alpine Linux, which will greatly reduce the image size.
- reduce the number of layers: this will also reduce the overall size of the image, especially if you use a multi-stage build (5.3.1) to only copy over the files you need.
- If you **get the container image under 50MB, you'll get a prize**!

7.2 Composing a backdoor.

TL;DR: our friend who's really into containerization has some diskpics on his server that we really need to see, so let's try to use the Flask app from Listing 17 to create a "backdoor" into his Docker host so we can syphon all his files. As a prank, of course...

The app can serve any files it has access to, but our friend'll surely want to run it in a container, so we'll inevitably need to trick them into bind-mounting their Host's root directory into the container as we've seen in Listing 26.

The challenge will be getting our friend to run it. They're so adept at Docker commands that they'll instantly notice if we ask them to run "docker run -v /:/realroot", so we'll need to figure out a way to hide the volume mount, as well as the ENTRYPOINT for our container.

You'll need to:

- take said Flask app (17) and its Dockefile (19) and modify it so that the ENTRYPOINT of the image is NOT anything suspicious like "python3 ...".
- write a Docker Compose file like in Listing 29 to:
  - define the container from our new "less-suss" image.
  - set up the mountpoints for the container.
  - ensure the networking settings will allow up to access the Flask app running in the container on the Docker host!
- Write the "docker-compose" command to deploy the compose file and send it to a friend IRL too!

Some quick tips:

- Rember that the image building stages from Dockefiles are simply run inside a container, and you have full control over all the files. Try to "backdoor" one of the existing system files (e.g. /bin/echo), or even modify some config options in the container so that it starts the Flask app in secret!
- make sure any bind-mounts on the container are Read Only so we don't tip our friend off.
- Do NOT just use "–net=host", since our friend will inevitably figure out we're doing something nasty. Only allow the networking features you need!