Docker containers

Agenda

What exactly is a container - a deep dive

1. Why Containers Are Ephemeral

A container is simply:

a Linux process + a temporary root filesystem.

The filesystem layers (image layers + writable layer) exist only while the container runs.

Key Point:

If the main container process exits → container stops → writable layer disappears → the container is gone.

This makes containers *stateless* and *ephemeral by design*.

---

2. Container = Just a Process on the Host

When you run:

```
docker run nginx
```

Docker does NOT start a virtual machine.

It launches a **Linux process on the host**, but inside kernel namespaces.

Diagram

```
Host OS
│
├── systemd (PID 1)
├── sshd
├── containerd
│     └── containerd-shim
│           └── nginx   <-- THIS IS THE CONTAINER!
```

The "container" is simply the **nginx process** running under isolation.

---

3. Why Container Isolation = Pseudo-Isolation

Namespaces hide parts of the world
Cgroups limit resource usage

BUT:

- All containers share the **same host kernel**
- All container processes appear in the host's `ps -ef`
- A root user inside the container **is not a real root** (User Namespace)
- A process inside a container can't break out *normally*, but under kernel vulnerabilities it **can escape**, because it's not a VM.

Thus:

Containers ≠ Virtual Machines

Containers = Regular Processes with illusions

That's why we call it **process-level isolation**, not OS-level isolation.

---

4. Understanding containerd, shim, runc

Modern container architecture:

```
docker --> containerd --> shim --> runc --> your process
```

containerd

- Long-running daemon
- Manages containers
- Handles images, snapshots, networking

containerd-shim

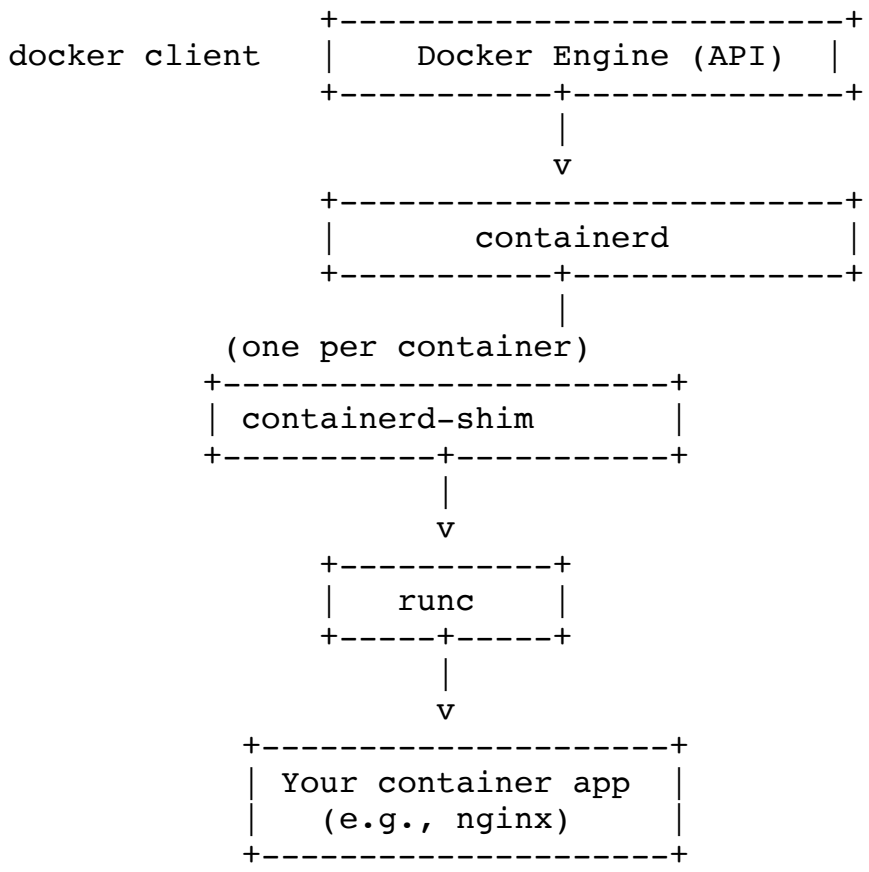
- One shim **per running container**
- Acts as parent for container process
- Allows containerd to restart without killing containers

runc

- The actual runtime

- Uses Linux kernel syscalls (`clone`, `unshare`, `pivot_root`)
- Sets up namespaces & cgroups
- Starts the main process inside the new isolated environment

Diagram



---

5. Process Tree Explains Container Behavior

Containers behave like this:

- A container is started by **containerd-shim**
- Shim launches **runc**
- runc sets up namespaces
- runc starts the container's *main process*, e.g., `/usr/sbin/nginx`

Process Tree Example

```
$ pstree -pl

containerd(233)
  └── containerd-shim(9421)
```

```
└── nginx(9422)
    ├── nginx worker(9423)
    └── nginx worker(9424)
```

IMPORTANT:

- The **main process inside the container** becomes **PID 1** *in the container's PID namespace*.
- But on the host, it is just another process (like 9422).

---

6. Why killing the main process stops the container

Because:

The container lifecycle = lifecycle of its PID 1

If PID 1 exits in the container namespace:

- containerd-shim detects process exit
- shim notifies containerd
- containerd marks container as stopped
- writable layer is discarded
- networking is removed
- cgroups cleaned
- container disappears

Example:

`docker kill <container>`

This sends SIGKILL to PID 1.

Container stops instantly.

---

7. Why containers are ephemeral (with diagrams)

Each container has:

- a **read-only image**
- a **temporary writable layer**

Diagram

```
Image Layers (read-only)
  layer 1
  layer 2
  layer 3
-------------------------
Container Writable Layer (deleted on stop)
  /var/log/app.log
  /tmp/files
-------------------------
Running Process
```

When container stops → writable layer is deleted → all state lost.

Thus:

Container data disappears unless stored in:

- Volumes
- Bind mounts
- External storage (DB, S3, etc.)

---

8. Why containers appear like isolated machines

Namespaces create illusions:

PID Namespace

Container process sees itself as PID 1.

Network Namespace

Container sees its own:

- eth0
- IP address
- routing table

Mount Namespace

Container sees its own root filesystem.

UTS Namespace

Container sees its own hostname.

IPC Namespace

Own shared memory.

User Namespace

Root inside container ≠ root on host.

---

9. Why containers start so fast

Because:

No kernel boot
No virtual hardware
No BIOS
No init system unless you add one

Containers start a single process:

runc → clone syscall → process runs

Start time: **~50ms**

VMs take: **tens of seconds**

---

10. Why containers are light

Containers share:

- The host kernel
- The host OS
- Libraries (optional)
- CPU scheduler
- Memory management

Only the filesystem + process isolation is unique.

---

11. Why containers fail if the main process exits

Because a container = **one main process**.

Examples:

- `nginx` dies → container dies
- `python app.py` exits → container exits
- `sleep 5` completes → container exits immediately

This confuses beginners.

Containers need:

- process managers (`supervisord`)
- or multi-process apps designed properly

---

12. Summary in 10 bullet points

1. A container is just a **process** + **isolation**
2. It runs on the **host OS**, not in a VM
3. Isolation is via **namespaces**
4. Resource limits via **cgroups**
5. Process started by **runc**, managed by **containerd-shim**
6. The container's root process is **PID 1** in its namespace
7. If PID 1 dies → container stops
8. Writable layer is temporary → ephemeral
9. Isolation is not strong like a VM → "pseudo-isolation"
10. All container processes visible in host `ps -ef`

CONTAINERS EXPLAINED — FROM ZERO TO DEEP INTERNALS

1. First: What EXACTLY is a container?

A **container is NOT**:

- a virtual machine
- a mini-OS
- something magical

A **container IS**:

A normal Linux process with special isolation applied using kernel features.

That's it.

Why does it *feel* like a separate computer?

Because:

- It has its own user space
- It has its own Inter process communicaton space
- It has its **own filesystem**
- Its **own network (space) interface**
- Its **own hostname**
- Its own **process tree**
- Its own **resource limits**

But none of this is done by creating a new OS.

All of it is done using:

✔ Linux Namespaces

✔ Linux Cgroups

✔ Chroot / Filesystem overlays

✔ Capabilities

✔ Seccomp (optional)

---

2. Why a container is just a process

Normally, when you run a process:

```
$ python app.py
```

the process shares:

- host network interfaces (eth0)
- host filesystem
- host PID number space
- host users
- host resources

In a container:

```
docker run python:3.11
```

Docker creates a new process:

– an instance of containerd-shim-runc-v2

python (inside container) → but still a Linux process on host

But before launching it, Docker enables isolation:

✔ PID namespace
✔ NET namespace
✔ UTS namespace
✔ MOUNT namespace
✔ USER namespace
✔ IPC namespace

Because of these namespaces, the process sees a **fake world**.

---

3. The Magic Behind Containers: Linux Namespaces

Think of namespaces like **AR (Augmented Reality) goggles**.

Each namespace puts a filter on what the process can see.

Let’s break down the namespaces.

3.1 PID Namespace (Process ID Isolation)

Without namespace:

```
Host Processes:
1 systemd
22 sshd
300 nginx
4212 python
```

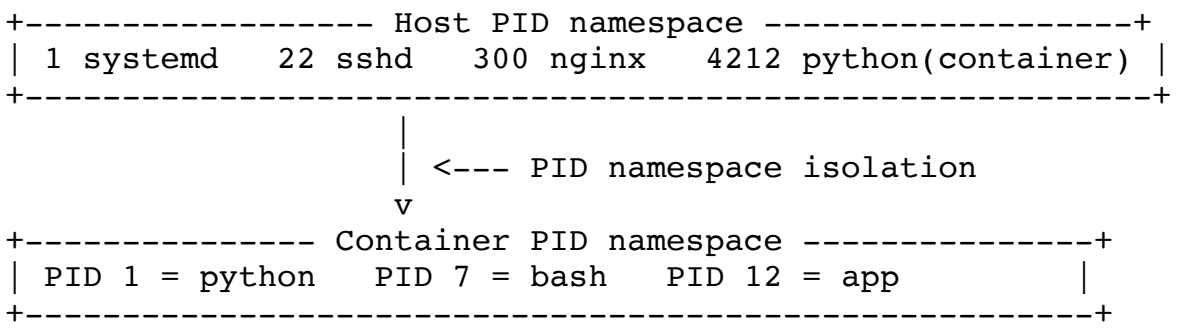
Inside a container:

```
/ # ps
PID   Command
1     python
7     bash
12    app
```

The container sees its own process as **PID 1**.

This makes the container feel like its own operating system.

Drawing:



3.2 Network Namespace

Each container gets its own:

- network stack
- IP address
- routing table
- firewall rules

But the host only sees a **veth pair**.

Drawing:

```
+---------------- Host ----------------+
| veth0 <======> docker0 bridge        |
+--------------------------------------+

Inside Container:
+---------------- Container -----------+
| eth0 -> 172.17.0.2                   |
+--------------------------------------+
```

This is why containers have **their own IPs**.

3.3 Mount Namespace (Filesystem Isolation)

Each container sees **its own root filesystem** (`/`).

But under the hood, Docker is doing:

- OverlayFS layers
- Chroot jail
- Bind mounts for volumes

```
Host filesystem:
/
├── bin
├── usr
└── home

Container filesystem:
/ (overlay)
├── bin
├── usr
└── app
    └── your files
```

Different root = **feels like a different machine**.

3.4 UTS Namespace (Hostname Isolation)

Lets a container have its own:

- hostname

- domain name

So inside container:

```
hostname = web-app
```

But host's hostname is:

```
hostname = ip-10-0-0-12
```

3.5 IPC Namespace (Shared Memory Isolation)

Containers get their own:

- semaphores
- shared memory segments

3.6 USER Namespace

Allows containers to run as:

- root **inside the container**
- but mapped to a non-root UID on the host

---

4. CGroups – Controlling Resources

Namespaces = isolation
Cgroups = resource limits

Cgroups limit:

- CPU
- Memory
- Disk I/O
- PIDs

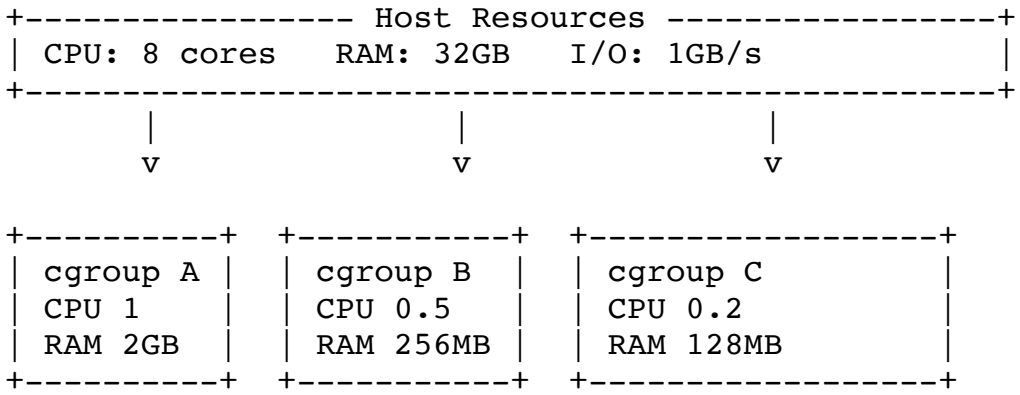
Example:

```
Limit container to 256MB RAM
Limit container to 0.5 CPU
```

If you run too many processes:

`cgroups prevents container from using more than allowed`

Drawing:



---

5. So How Does Docker Create a Container?

Step-by-step:

1. Pull an image
2. Unpack filesystem layers (OverlayFS)
3. Create cgroups for limits
4. Create namespaces
5. `chroot` into the new root filesystem
6. Start the process (e.g., python)

So docker run:

`docker run nginx`

is actually:

```
create-net-namespace
create-mount-namespace
create-pid-namespace
apply-cgroups
chroot to overlayfs
launch /usr/sbin/nginx
```

---

6. Why your normal programs DO NOT feel like containers

Your normal programs:

- share host network
- share host filesystem
- share host hostname
- share host PID tree
- share all CPU/RAM

Containers **hide all of that**.

---

7. VM vs Container (Deep Difference)

VM Architecture

```
Hardware
   │
Hypervisor
   │
Guest OS (Linux/Windows)
   │
Your App
```

Container Architecture

```
Hardware
   │
Linux Kernel
   │
Your App (isolated using namespaces + cgroups)
```

Docker does NOT run a separate kernel → only processes.