
Containers vs. VMs

Key takeaways

1. A VM brings its own kernel; a container borrows the host's. Start time, memory, what can run inside, and what a bug exposes all follow from that.
2. Containers exist because the OS isolates processes, while an application is many processes sharing files, ports, and memory. They make the group one unit.
3. The VM is the tenant boundary (whose bill, whose kernel); the container is the deployment boundary (whose code). Firecracker and gVisor each borrow from the other.

Common misunderstandings of virtualization

Which claim would you defend right now? Choose before the evidence.

“Containers run code faster than VMs because they have less overhead.”

“A container is safe because it cannot see the host’s files.”

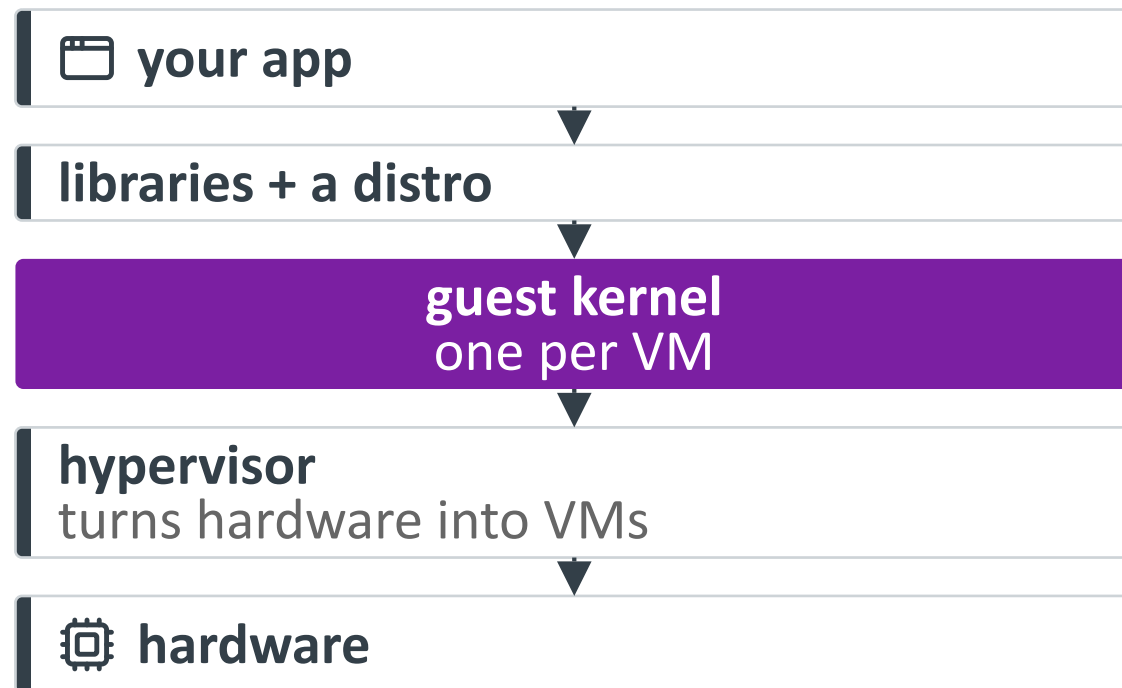
“Docker on a Mac runs Linux containers on macOS.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

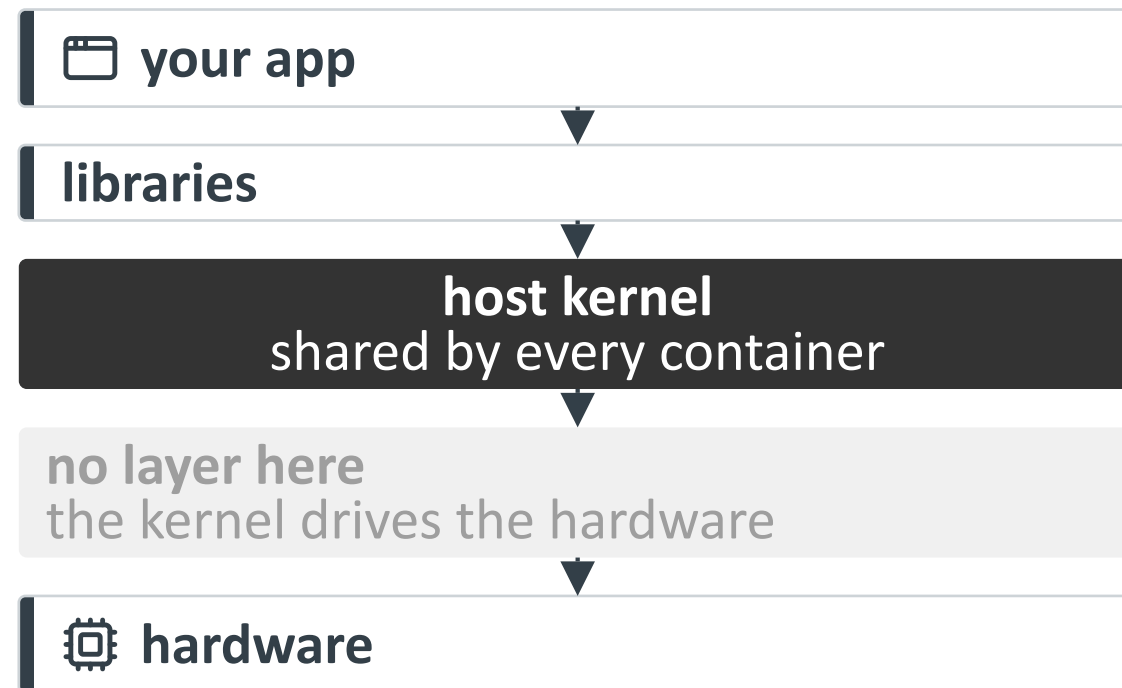
One difference: who supplies the kernel

Each arrow reads “runs on”; compare the stacks row for row

Virtual machine



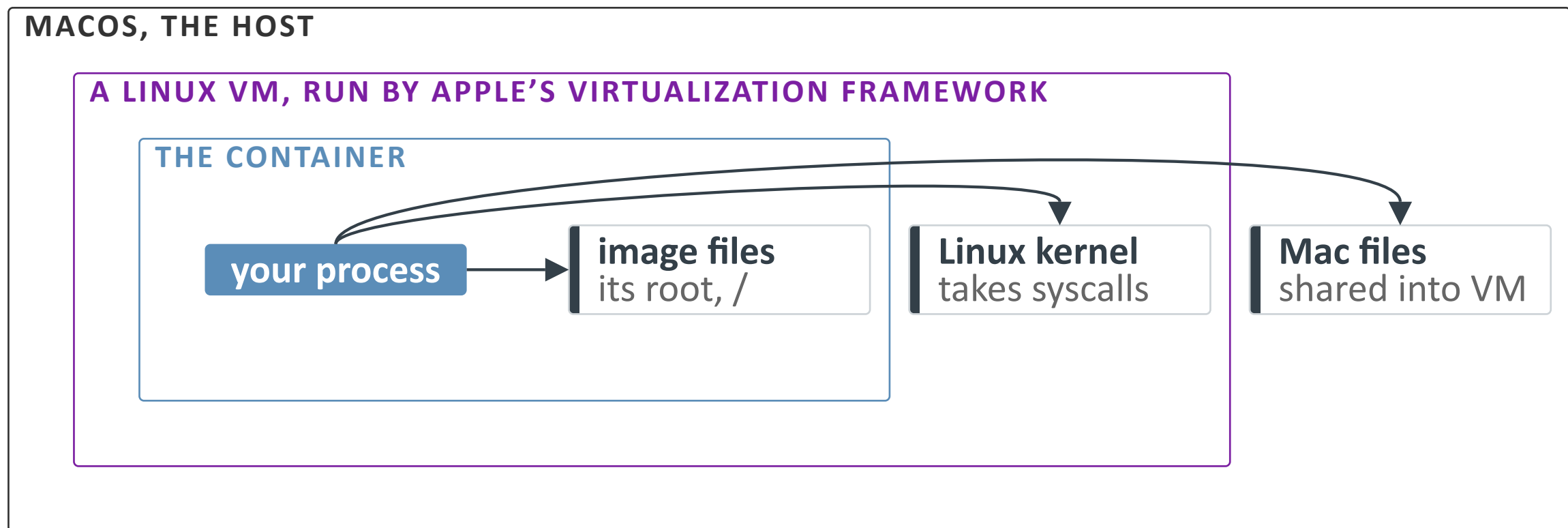
Container



Every other difference in this deck follows from that one row.

Docker on a Mac hides a Linux VM

The kernel's system-call interface is the contract: a Linux binary needs a Linux kernel



A VM can run any OS; a container runs only on the kernel it was built for

A container carries no kernel, so on a Mac the Linux kernel has to come from a VM.

A VM boots an OS; a container starts a process

FIVE STAGES Boot a VM

- 1 Firmware**
initializes the virtual devices
- 2 Guest kernel**
boots and probes those devices
- 3 init**
the first process, usually systemd
- 4 Services**
network, logging, ssh
- 5 Your app**
an EC2 VM: usable in roughly 10–40 s

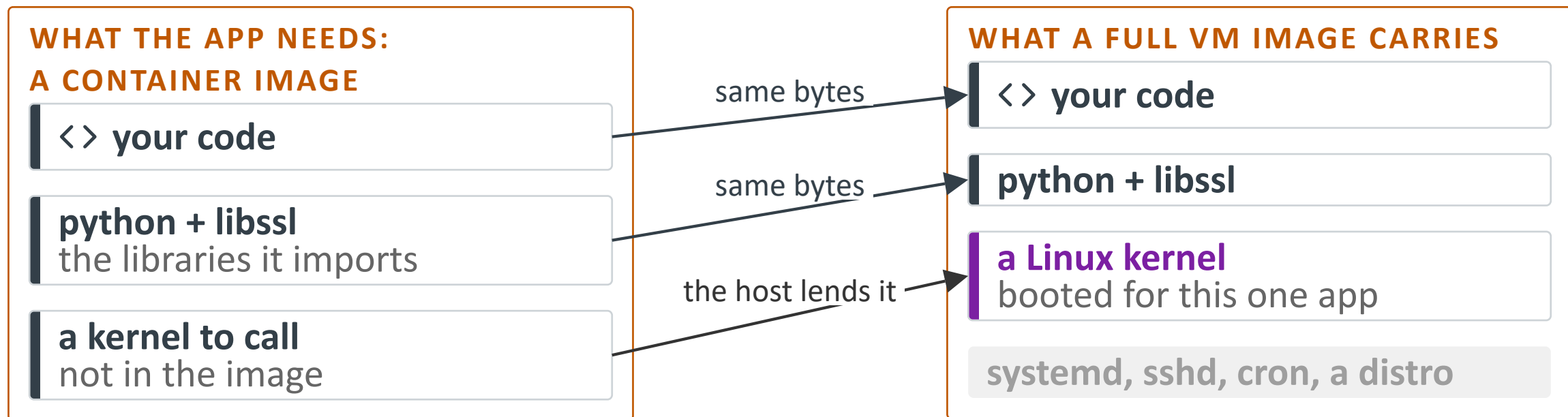
THREE STAGES, IMAGE ALREADY PULLED Start a container

- 1 clone into new namespaces**
a new process with its own views
- 2 Mount the image**
its layers become the process's /
- 3 exec your app**
running in tens to hundreds of ms

The container skips every step that brings up a kernel: one is already running.

Most of a VM is luggage the app never calls

Linked rows are what the app calls; the gray row is luggage that only the VM image ships



Container images, uncompressed: alpine 8.7 MB, python:3.12-slim 132 MB

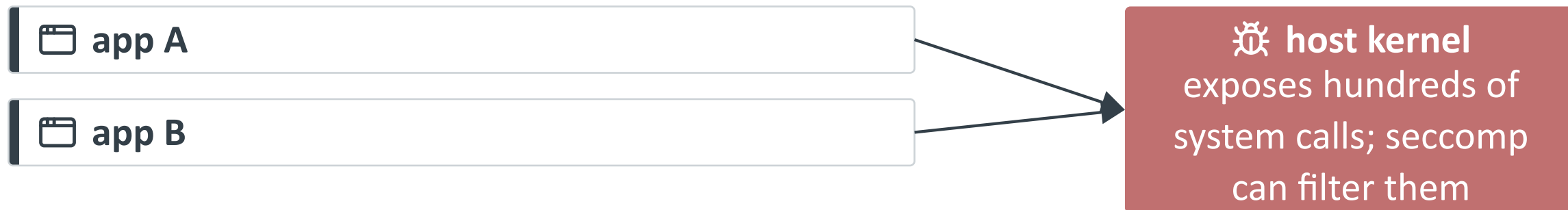
Amazon Linux 2023 VM root volume: 2 GB minimal, 8 GB standard

Containers won the deployment unit by shipping only what the app calls.

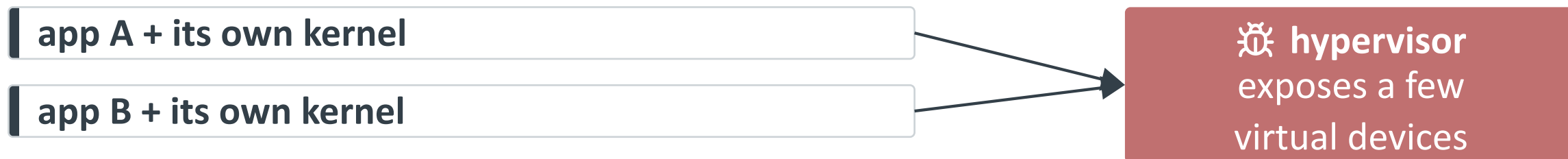
Isolation is as strong as the interface is narrow

Red marks the one shared layer: a bug there reaches every guest on the machine

CONTAINERS PUSH ON THE HOST KERNEL



VIRTUAL MACHINES PUSH ON THE HYPERVISOR



A kernel bug is a container escape; a VM escape needs a hypervisor bug.

Containers fix the OS's missing abstraction

The OS isolated processes, but an application is a group of them, and the group had no name

BEFORE

Three apps, one shared view

one pid list, one port 80 to fight over, one memory pool any app can drain

namespaces + cgroups

AFTER

Three accountable units

each sees itself as pid 1 with its own :80, and a cgroup caps its memory and CPU

- ✓ one kernel underneath all three
- ✓ one scheduler, now weighing groups

Inside the group the developer decides: the image, its libraries, its processes

Around the group the operator decides: the memory cap, the CPU share, which ports reach it

An application is a group of processes; a container makes the group the unit.

VMs sell what containers cannot

Not speed: the processor runs a VM's instructions directly, so CPU-bound code runs near native

Compatibility run any OS, unmodified: Windows next to Linux, an old distro next to a new one

Accounting sell a fixed slice of a machine and cap it, kernel and all

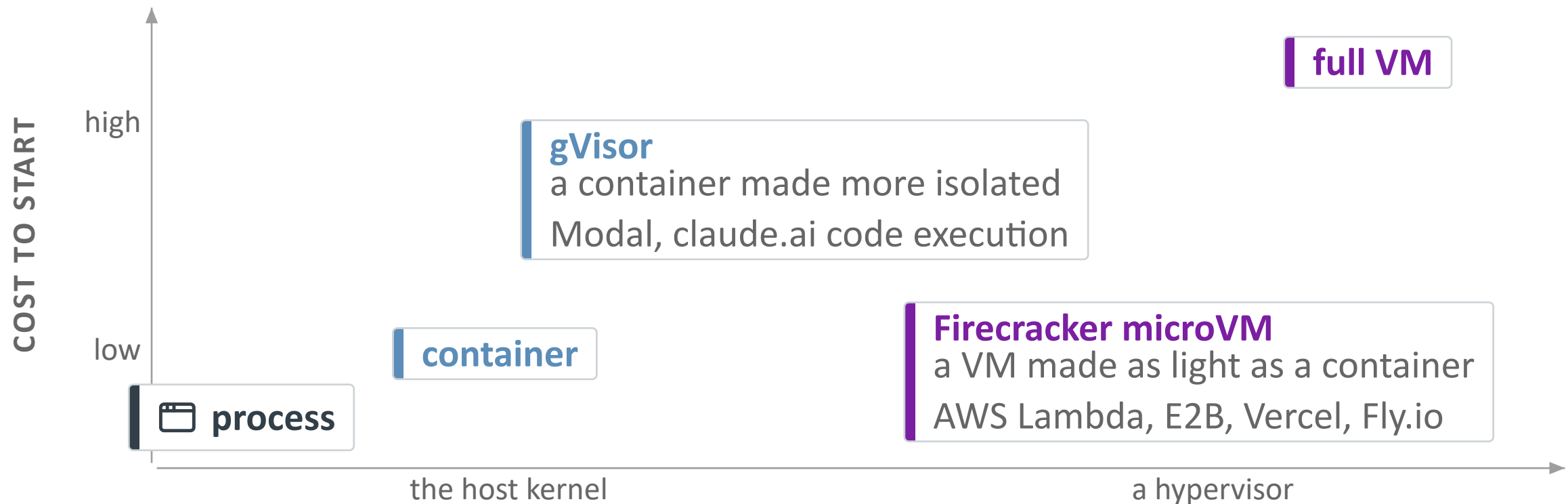
Migration move a running machine to another host without restarting it

Security untrusted tenants push only on a narrow virtual-hardware interface

Containers are not faster at running code; they are faster at starting it and denser to pack.

Each side is reaching for the other's strength

Positions are qualitative; blue descends from containers, purple from VMs

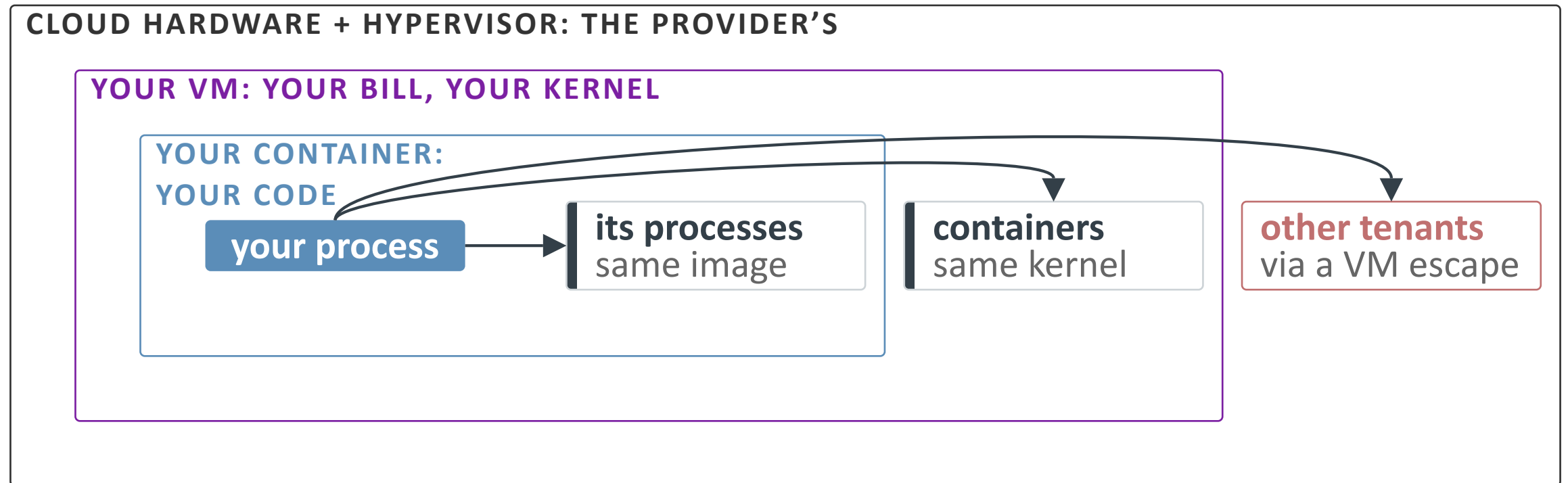


ISOLATION: WHAT A GUEST'S BUG REACHES

Both abstractions persist because each gap is being closed from both sides.

Tenant in a VM, deployment in containers

A cloud gives you both. Each ring names its owner; each card shares that ring with your process



The one ring shared with strangers is the one the provider owns

Your bill buys the VM; your deployment fills it with containers.

Key Takeaways

1. A VM brings its own kernel and a container borrows the host's. Start time, memory, what can run inside, and what a bug exposes all follow from that one difference, so ask which kernel your code needs before you pick a boundary.
2. Containers exist because the OS isolates processes while an application is a group of them sharing files, ports, and memory. Namespaces give the group its own view and cgroups cap what it uses; the kernel underneath stays shared.
3. The VM is the tenant boundary and the container is the deployment boundary, so a cloud gives you both. Put code you do not trust behind a narrow interface: a VM, a Firecracker microVM when it must start fast, or gVisor when it must stay a container.