
Containers and VMs in the wild

Key takeaways

1. What you ship to a cloud is an image, not code: a stack of filesystem layers or a frozen boot disk that the platform starts many times.
2. Every boundary has an owner: you own the image, the platform the sandbox, the cloud the VM and hardware. State survives only where its owner says it does.
3. “Serverless” is someone else booting a microVM for you, and code a model wrote is untrusted code: choose its boundary as a provider would for a stranger.

Common misunderstandings of the cloud

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

“Deploying to the cloud means uploading your code.”

“A container image is a paused container you can copy.”

“Serverless means there is no server and no virtual machine involved.”

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

The unit you ship is an image

Two artifacts, one role: an immutable thing the platform starts many times

WHAT DOCKER PULLS AND RUNS Container image (OCI)

manifest + config

names every layer by digest

↓ points to, by digest

layer archives

each a tarball of a directory diff

↓ stored in

a registry

serves blobs by sha256 digest

WHAT EC2 BOOTS Machine image (AMI)

AMI

Region, OS, architecture, root device

↓ points to

EBS snapshot

a frozen copy of the boot disk

↓ lives in

one Region

copy the AMI to use it elsewhere

You never ship the running thing: each launch starts a fresh copy of the image.

An image is a stack of directories

Each Dockerfile line adds a read-only layer; OverlayFS shows the stack as one root



WHAT THE STACK BUYS, AND WHERE IT BREAKS

 **Two images, one base**
both FROM python:3.12-slim: 132 MB on disk, stored once

rm hides, it does not shrink
a later layer masks it; the bytes still ship

The top dies with the container
the next one starts empty

Layers are shared and read-only; a container's own writes sit on top and die with it.

Modal makes the image a value

You define the image in Python; Modal builds it on first use and reuses it after

WHAT THE PLATFORM DOES WITH IT

```
app = modal.App("report")
image = (
    modal.Image.debian_slim()
    .pip_install("pandas==2.2.0", "numpy")
)
```

```
@app.function(image=image, gpu="A100")
def summarize(path: str) -> str: ...
```

 **Cached per layer**
change one; later layers rebuild

 **Boots in about a second**
plus imports and model loading

 **Scales to zero**
after scaledown_window idles

Billed per second
isolated with gVisor

Same image definition, same cached layers: the build is memoized like a pure function.

Four boundaries, and you own one

Each ring names what the boundary is made of and who runs it

CLOUD HARDWARE — THE CLOUD PROVIDER'S

A VM — MODAL RENTS IT

GVISOR SANDBOX — MODAL RUNS IT

YOUR IMAGE — YOURS

<> **your function**
the only code you wrote

WHERE YOUR STATE LIVES

The container's files
gone when it scales down

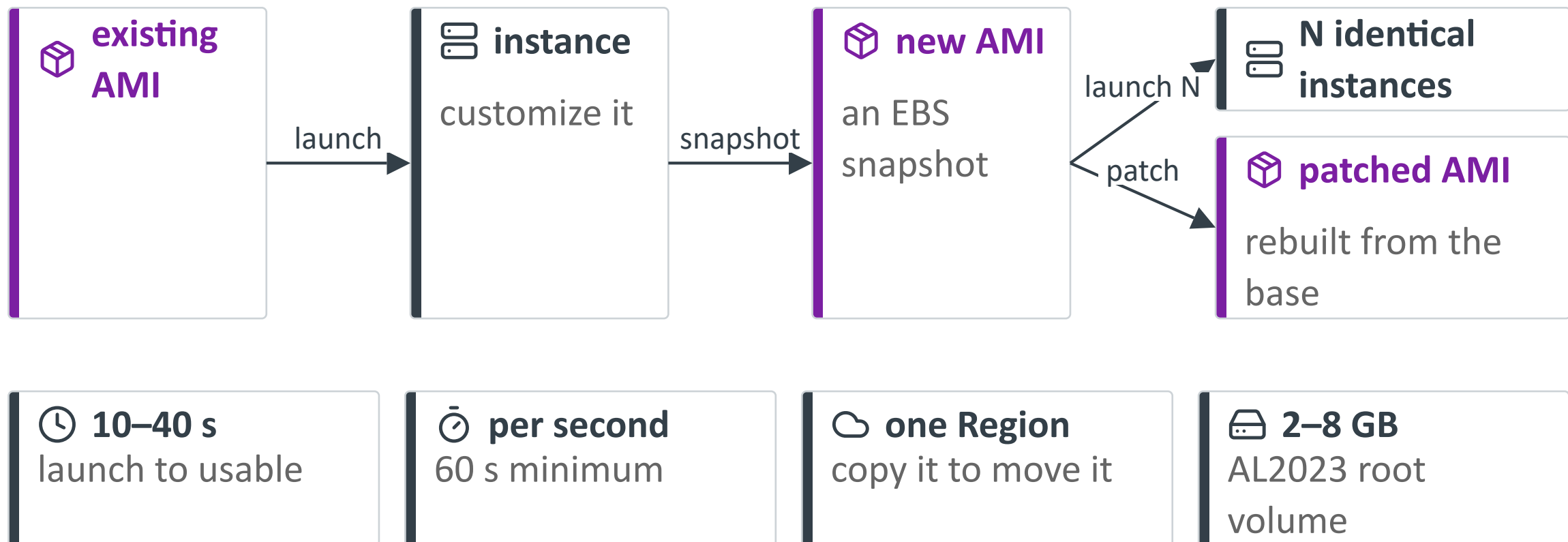
 **modal.Volume**
outside the sandbox; commit() to
share writes

Where the rings change
a Modal Sandbox can trade gVisor
for a VM with its own kernel

The boundary you cannot cross is the one someone else owns.

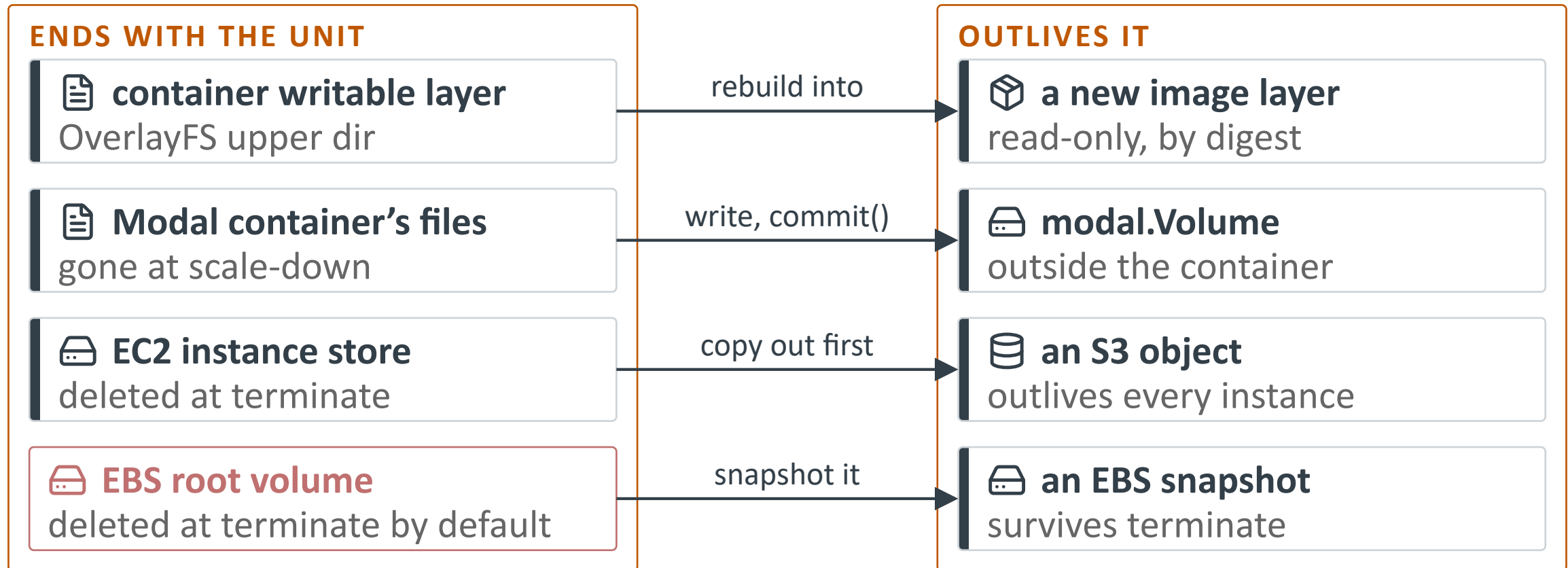
An AMI is a frozen boot disk

Build servers by stamping copies, never by editing a running one



A fix is a new image: replace the old instances, never repair them in place.

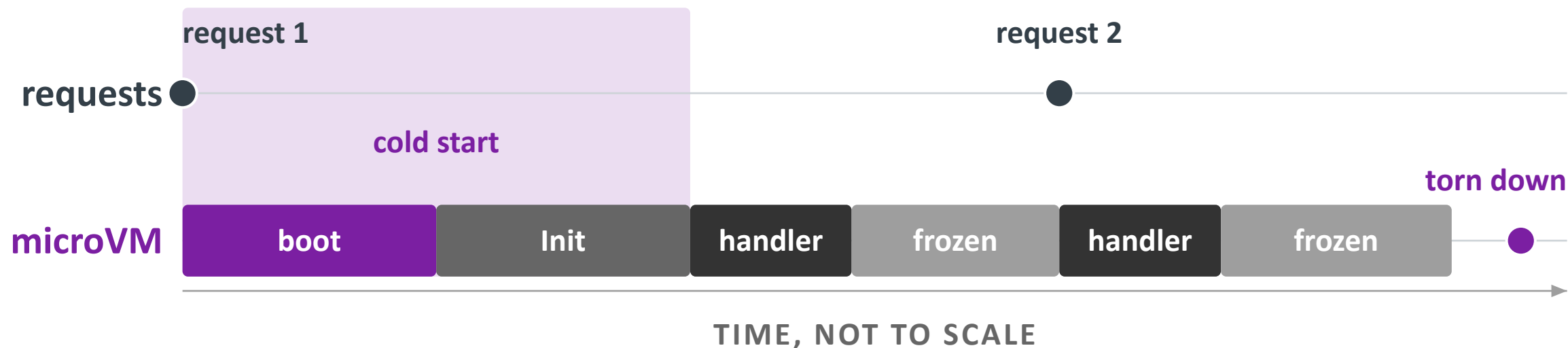
Durability belongs to the volume



Ask who promises to keep a byte; the running instance's disk makes no promise.

The cold start you feel is a VM booting

AWS Lambda runs each function in a Firecracker microVM, then freezes it between requests



⚡ **<125 ms**
to guest boot,
best case

🕒 **≈350 ms**
end to end,
measured

💻 **<5 MB**
memory per
microVM

<> ≈50k lines
of Rust; five devices

“Serverless” is a VM someone else boots; a cold start is a request waiting for it.

Model-written code is a stranger's code

AS EACH VENDOR DOCUMENTS IT	ROW COLOR: WHAT THE BOUNDARY IS MADE OF		
	the stranger	 bug reaches	start cost
Claude Code, local Seatbelt, bwrap	the agent, as you	your kernel	none: a process
claude.ai gVisor	anyone's chat	gVisor's kernel	a container start
Modal Sandboxes gVisor; VM opt-in	your users, agents	gVisor's kernel	about a second
E2B, Vercel, Fly.io, Lambda Firecracker	any customer	a guest kernel	≈350 ms

Give generated code the boundary a cloud gives a stranger: a sandbox kernel or a microVM.

Key Takeaways

1. What you ship is an image: an OCI manifest naming layer archives by digest, or an AMI that points at a snapshot of a boot disk. The platform starts many copies, so fix a bug by building a new image, never by editing a running one.
2. Every boundary has an owner, and state survives only where its owner promises: a container's writable layer and an instance's disks end with the unit; a volume, a snapshot, or an object store keeps it. Put state there on purpose.
3. Serverless boots a microVM for you, and that boot is the cold start you feel. Code a model wrote is a stranger's code: run it behind the boundary a provider gives a stranger, a gVisor sandbox or a microVM, not your own shell.