WORKSHOP · PART ONE

# Fencing the yard

## Why your dev environment needs a sandbox — and how to pick one

## What you'll leave with

- **Why** a developer laptop is now one of the most valuable targets in a company.
- **Four questions** to ask any sandbox: about files, credentials, network and control.
- **A checklist** for comparing tools, so you can choose one that fits *your* risks.
- Then, hands-on: one of those tools running your own project.

> *You don't have to pick the tool we use today. But you should leave being able to choose a sandbox that works for you.*

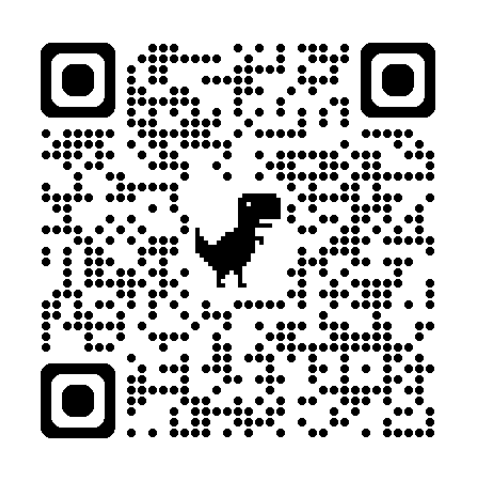
## Setting up (1/2)

If you're on Windows you have to use WSL2: in PowerShell **as Administrator**, `wsl --install -d Ubuntu-24.04`, then reboot when asked.

Otherwise, these are the tools we'll need to install first:

- `uv`, a Python runtime and package manager
- `lima`, or Linux Machines, the virtual machine manager
- `podman`, an open source, daemonless alternative to Docker with safer defaults

Find all the commands on: https://twistco.github.io/foldyard/workshop/#before-the-day



## Setting up (2/2): in a terminal

```
# macOS - if you don't have Homebrew, see how to install at https://brew.sh/
brew install uv lima podman

# Linux / WSL2 (Debian/Ubuntu; x86-64 with KVM) — plus the KVM and WSL steps on the workshop page
sudo apt-get update && sudo apt-get install curl qemu-system-x86 qemu-utils ovmf podman
curl -LsSf https://astral.sh/uv/install.sh | sh    # then open a new terminal
curl -fsSL https://github.com/lima-vm/lima/releases/download/v2.2.0/lima-2.2.0-Linux-x86_64.tar.gz \
  | sudo tar Cxz /usr/local
```

Then install the sandbox `foldyard` itself and start downloading the VM image:

```
uv tool install foldyard
foldyard prereq --fetch
```

PART 1

# Your laptop is the prize

## What's on your laptop right now

- SSH keys — push access to every repo you can reach
- Cloud logins: `gcloud`, `aws`, `az` — often with admin rights
- GitHub / GitLab tokens, `~/.npmrc`, `~/.pypirc`
- `.env` files with real API keys
- Browser sessions, password-manager sessions
- **Every other project** you've ever cloned
- Your AI tools' logins and settings
- Your editor and its extensions
- Your shell history

All of it readable by **any program you run** — because it runs as *you*.

## Who actually runs code on your laptop?

- **You.** Hopefully carefully and not permanently as admin.
- **Every dependency you install.** `npm install`, `pip install`, `uv sync` run install scripts — code from strangers, with your rights, before you've read a line of it.
- **Your build tools and dev servers**, and everything *they* pull in.
- **Your editor's extensions**, updated automatically.
- **AI agents**, which read untrusted text (issues, web pages, docs, other people's code) and then run commands.

> *The shared entry point: **code you didn't write, running with your rights.***

## Story 1 · A worm in the package registry

**Shai-Hulud** — npm, September 2025, and again since.

1. You install a package that a compromised maintainer account republished.
2. Its install script searches your laptop for secrets (it literally runs a secret scanner).
3. It uses your npm token to publish infected versions of **your** packages.
4. It dumps what it found into a new public GitHub repo.

The worm spreads itself. Its fourth generation (2026) also targets CI pipelines and plants persistence in developer tools.

Sources: Krebs on Security (Sep 2025) · ReversingLabs (Sep 2025) · Tenable, Mini Shai-Hulud FAQ (2026)

## Story 2 · Malware that asked your AI for help

**Nx "s1ngularity"** — August 2025.

- Malicious versions of a popular build tool shipped an install script.
- It looked for AI command-line tools on the laptop — Claude, Gemini, Q — and ran them, safety prompts off, to **search the disk**.
- Results went to a public repo in the victim's own GitHub account.
- In a second wave, stolen tokens were used to make thousands of private repos public.

Your AI tools are capable, logged in and *on your laptop*. That makes them part of the attack surface.

Sources: Wiz · StepSecurity (Aug 2025)

## Story 3 · Instructions hidden for your agent

**TrapDoor** — May 2026, npm + PyPI + crates.io.

- The usual: packages that steal SSH keys, cloud credentials and tokens.
- The new part: pull requests to popular AI projects that try to add `CLAUDE.md` and `.cursorrules` files with hidden instructions.
- The instructions asked the coding assistant to run a "security scan" — that finds and sends out secrets.

A file your agent reads *is* code your agent runs.

Source: Socket research, reported by The Hacker News (25 May 2026)

## Story 4 · The editor update

**A trojanised VS Code extension** — May 2026.

- A backdoored release of a popular extension was live on the marketplace only for **minutes**.
- Auto-update delivered it to existing users as a routine update.
- One of the affected laptops belonged to an employee at GitHub; GitHub confirmed internal repositories were taken.

Nobody clicked anything suspicious. The update *was* the attack.

Sources: BleepingComputer · Cloud Security Alliance research note (May 2026)

## Story 5 · It comes back after you clean up

The 2026 worm generation installs **persistence** on developer machines:

- a background daemon that watches your GitHub token — and deletes files if you revoke it;
- injected **editor tasks** ( `.vscode/tasks.json` );
- injected **AI-agent hooks** in `~/.claude/settings.json` .

The published advice: find and remove all of it *before* rotating your credentials.

> *Rotating keys after an infection is a must, yes, it's annoying*

Source: Tenable, Mini Shai-Hulud FAQ (2026)

## Story 6 · The key nobody revoked

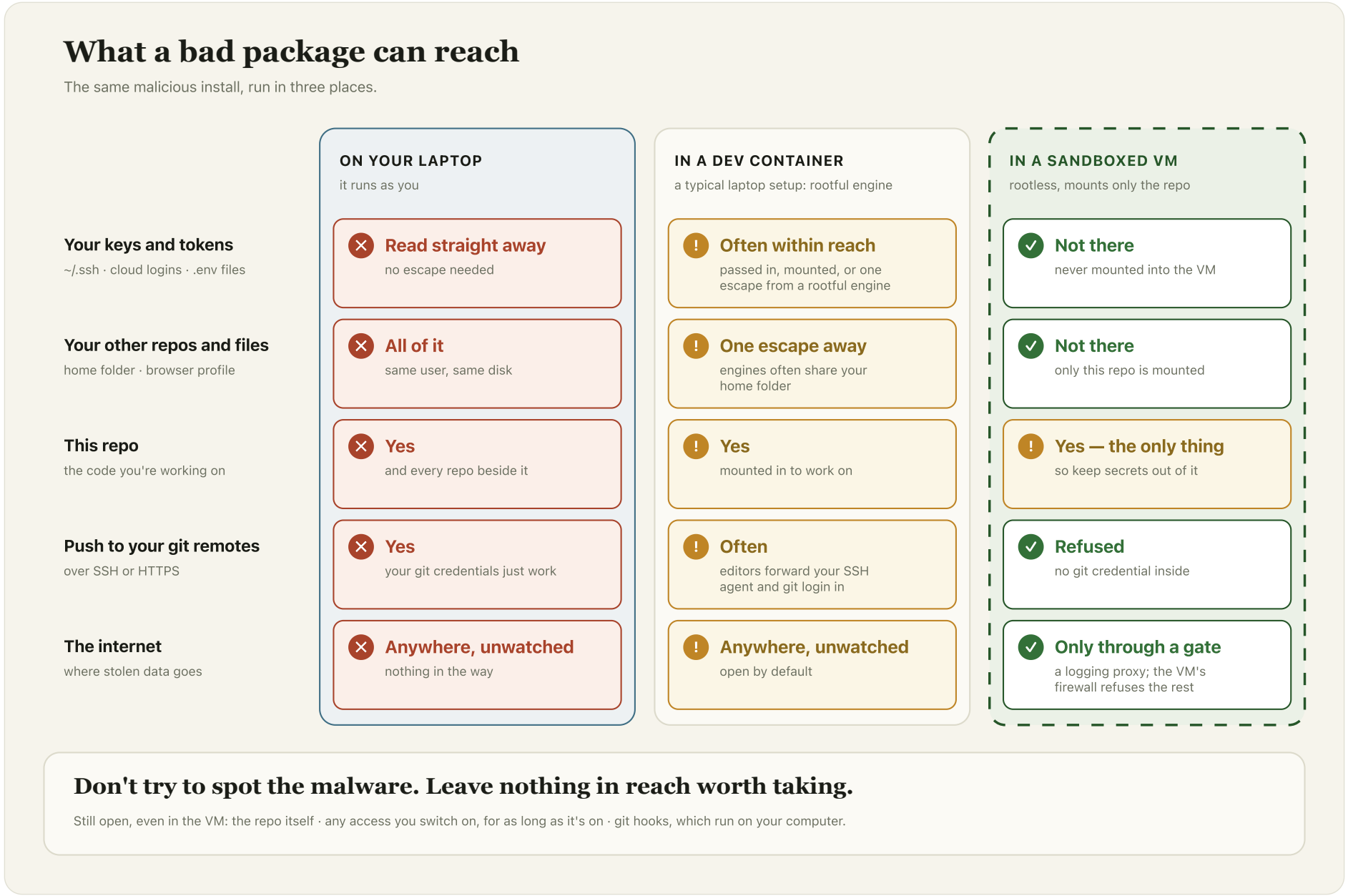
**LiteLLM** — the AI gateway, October 2026. Its second supply-chain compromise this year.

1. **May:** a coding agent's config file ( `opencode.json` ) holding a platform master key is pushed to a public repo. The commit is force-pushed away. The key is **never revoked**.
2. **October:** the key fetches an org owner's GitHub CLI token from that platform. The token covers the whole account, not scoped down.
3. With it: CI secrets sent out, two maintainers' tokens, a release built from a lookalike fork.
4. A backdoored `litellm` is live on PyPI and Docker Hub for about two hours.

> *Hiding a leaked key isn't revoking it.*

Sources: LiteLLM, attack chain (23 Oct 2026) · Datadog Security Labs · Socket

# The pattern: everything is one hop away



# The lethal trifecta

Simon Willison's name for the combination that turns an AI agent into a data leak:

### 1 · Private data

Your keys, your code, your customers' data.

### 2 · Untrusted content

Issues, web pages, docs, dependencies — anything someone else wrote.

### 3 · A way to send data out

Network access — or anything that can *become* network access (a push, a comment, a URL).

A developer laptop running an agent has **all three by default**. Take away **any one** and the attack falls apart. A sandbox is a way of taking away legs.

Source: Simon Willison, The lethal trifecta for AI agents (16 June 2025)

PART 2

# What a sandbox must get right

## Four questions — then one more

| | Ask | What "good" looks like |
|---|---|---|
| **Files** | What can code inside *see*? | Only the project. Not your home folder, not your other repos. |
| **Credentials** | What can it *hold*? | Nothing by default. Access you switch on, for a while, narrowly. |
| **Network** | What can it *send*, and where? | Everything goes through a gate you control, and the gate can't be skipped. |
| **Control** | Can the inside *move the fence*? | No. Config the inside can edit doesn't take effect until you approve it. |

**And:** can you **check** it? A fence you can't test is a fence you're taking on trust.

## How strong is the fence?

| Fence | Separates | Typical gap |
|---|---|---|
| **Permission prompts** in the agent | intentions | You click "yes" 200 times a day. Install scripts never ask. |
| **Process sandbox** (macOS Seatbelt, Linux Landlock / bubblewrap) | one process | Usually one agent, not your whole stack; rules are easy to get subtly wrong. |
| **Container** | processes, files | Shares your kernel. Whoever has the container engine's socket is effectively root. |
| **Virtual machine** | a whole OS | Its own kernel. Costs memory and some speed. |
| **Remote machine** (Codespaces and friends) | everything | Your code and data live on someone else's computer; you need a network. |

Stronger fences cost more. The question is what you need to keep *in*, and what you need to keep *out*.

# The landscape

## Where foldyard sits

Ask any dev sandbox two things: how strong is its fence, and what's inside it? Then ask where the credentials live.

HOW STRONG THE FENCE IS →

| | one agent run<br>or one task | one project's toolchain<br>a single dev container | the whole dev stack<br>services, installs, the agent |
|---|---|---|---|
| **VM**<br>its own kernel | ● **Per-task microVMs**<br>gondolin · matchlock · microsandbox<br>a proxy adds credentials on the way out; one agent or task each | ○ **Cloud workspaces**<br>Codespaces · Gitpod · Coder: a dev container in a VM or container on someone else's computer | ◉ **foldyard**<br>the stack, the dev box and the agent in one VM per project, on your computer; credentials stay outside, added on the way out<br>● **Docker Sandboxes**<br>a microVM per agent with its own Docker engine: compose runs inside<br>○ **A VM per worktree**<br>vmoat (early): no secrets story |
| **container**<br>shares a kernel | ○ **Agent in a container**<br>container-use · vhrn | ○ **Dev containers on a laptop**<br>rootful engine; secrets come in as env vars or files | ○ **Compose on a laptop**<br>the services are fenced; installs and the agent usually run as you |
| **process sandbox**<br>shares your OS | ○ **Agent process sandboxes**<br>nono · sandbox-runtime · Coder Boundary | | |

WHAT'S INSIDE THE FENCE →

### Reading it

**Up: a stronger fence**
A process sandbox shares your OS, a container shares a kernel, a VM brings its own.

**Right: more inside it**
Fence one agent run, and your installs, dev servers and stack still run outside it, as you.

**The dot: credentials**

- ● kept outside the fence, added on the way out
- ○ inside the fence, or not handled at all

Every piece exists somewhere. Two tools reach the top-right corner: Docker Sandboxes (per agent, closed source) and foldyard (per project, open source, timed credentials).

*Placed by category, from prior-art.md (2026-06 to 10). Check a tool's docs before you rely on a placement.*

## Files: mount the project, nothing else

- A **container engine on macOS or Windows** runs inside a VM — and that VM often shares your **whole home folder** into it. One container escape and every key is readable again.
- "Mounts only the repo" also means the repo's **ignored files**: `.env`, `*.pem`, `credentials.json`.
- So: **scan the working tree, not just history** — then move real secrets out of the folder.

```
gitleaks git .        # the history
gitleaks dir .        # every file in the folder, ignored ones too
```

> *A sandbox can mount a clean repo only. It can't clean a dirty one.*

## Credentials: the best secret in a sandbox is none

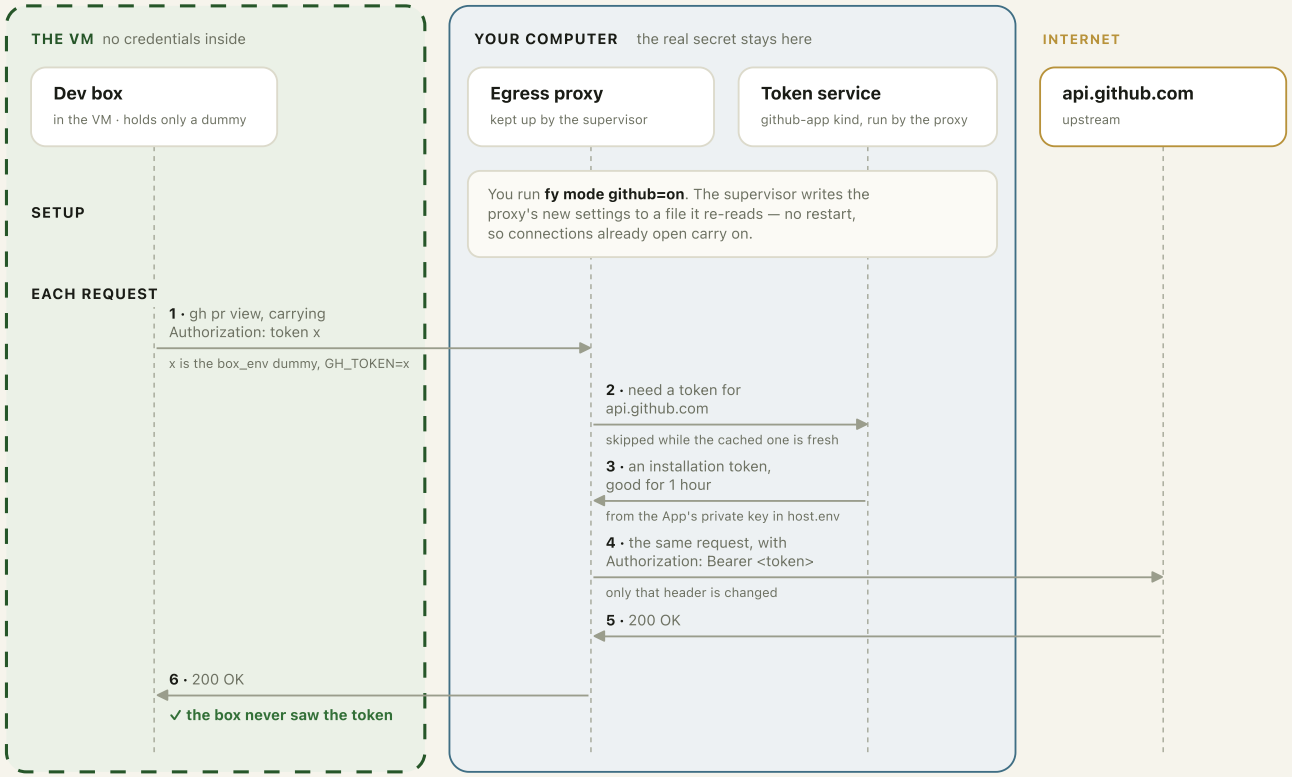
- **Nothing at rest.** No key files, no token variables, no credential helpers inside.
- **Swap at the edge.** The inside holds a dummy; a proxy *outside* swaps in the real credential as the request leaves. The API sees the real token; the sandbox never does.
- **Short-lived and narrow.** Prefer tokens that expire in an hour and can do one job.
- **"Act as me" is an emergency.** Anything with *your* identity gets a timer and switches itself off.

This used to be rare. In 2026 it became close to standard in agent sandboxes — good news when you're choosing.

# Swapping the credential at the edge

## Credential injection — the token never enters the box

The box sends a dummy. The proxy on your computer swaps in a real token on the way out, and answers for itself when it can't.




### Why it holds

**1 · Nothing to steal**

The real token lives only in the proxy on your computer and on the wire to GitHub. A compromised box can use it while the switch is on — never read it, or take it away.

**2 · Pinned to one host**

The GitHub kinds inject only on api.github.com, so the token is never aimed at another host. A git push to github.com carries no token.

**3 · Same swap for agents**

Keyless Claude and Codex work the same way: a dummy key in the box, your real one added at the provider's API host.

**4 · Switching cuts nothing**

The proxy re-reads its settings, so turning a switch on or off never drops a download or an agent's stream mid-flight.

The switch, in foldyard.toml:

```
[[inject]]
switch  = "github"
kind    = "github-app"
app_id  = "123"
installation_id = "456"
box_env = {GH_TOKEN="x"}
```

The App's private key goes in host.env, never in this file.

fy verify checks that every dummy in the box still holds its dummy value — a real token there is the leak it looks for.

### WHEN IT CAN'T INJECT, IT SAYS WHY

the proxy answers the box itself instead of sending the dummy upstream

**Switch off**
`401 from the proxy`

A request carrying the dummy is answered, not sent. The message names the fix, fy mode github=on, to run on your computer — and gh prints it.

**Token service failed**
`502 from the proxy`

"foldyard could not mint a credential for ..." with its reason. Remembered for 15 s, so retries don't run it again. Fixed on your computer: fy host logs.

**GitHub says 401**
`new token, resend once`

The cached token was refused, so a fresh one is made and the request sent once more. If none can be made, the box gets the 502 instead.

**Two switches, one host**
`502 · no credential named`

Allowed if each writes its own header: the dummy a request carries picks the credential, and neither or both is refused. Same header? fy mode refuses the second.

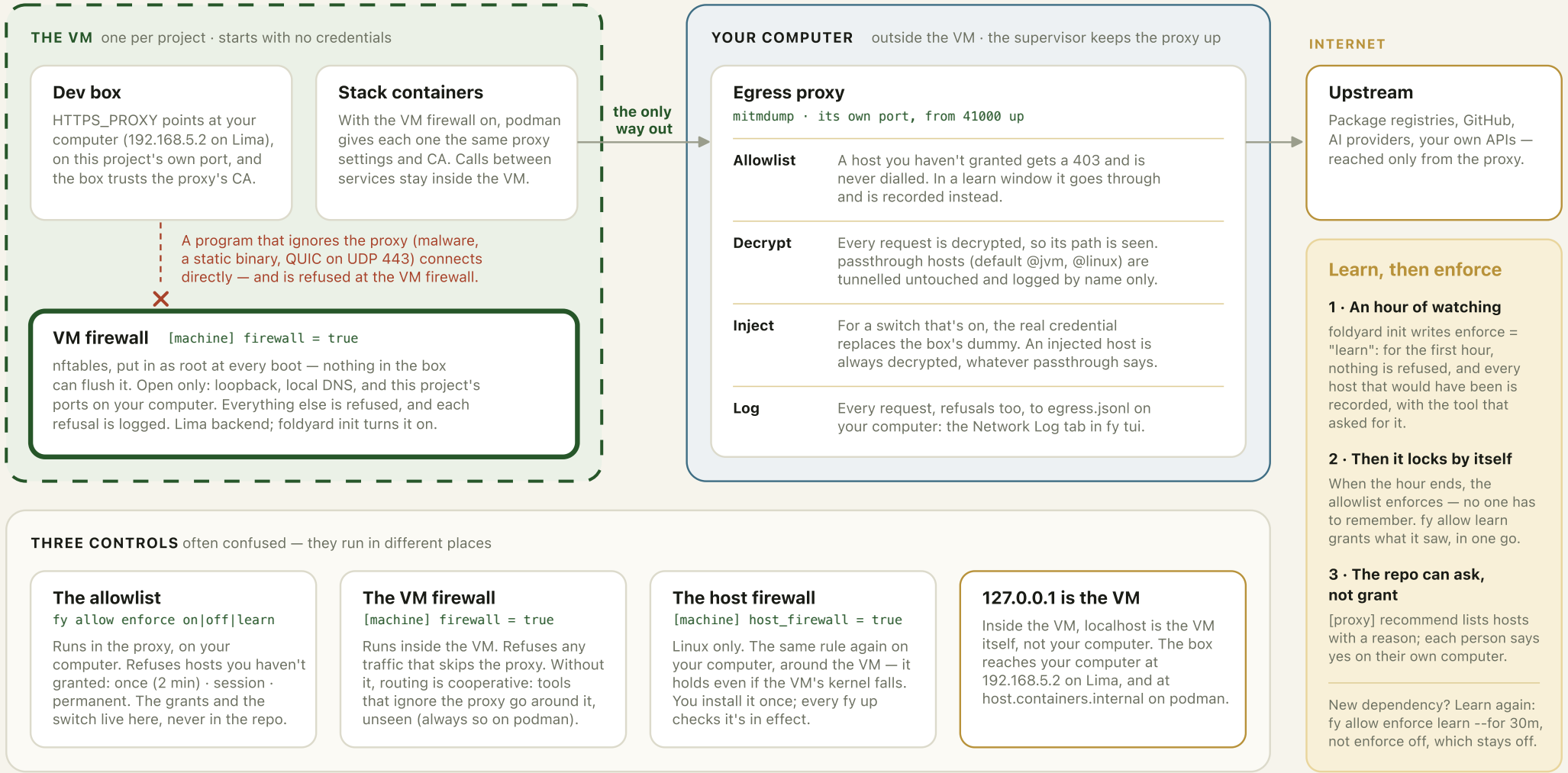
## Network: the third leg

- Exfiltration needs a way out. **Controlling egress is how you take away that leg.**
- **Proxy settings are a suggestion.** Well-behaved tools honour `HTTPS_PROXY`; malware, Go binaries and many SDKs ignore it.
- **A firewall makes it a rule.** Traffic that skips the proxy is refused, not just logged.
- **An allowlist** decides where traffic may go. Learn for a while, review what you saw, then enforce.
- Watch for **"trusted" hosts** that are also places anyone can upload to (package registries, GitHub, pastebins).

# Gating the way out

## How traffic leaves the box

One way out: a proxy on your computer that logs, decrypts, allowlists and adds credentials. The VM firewall refuses the rest.



## Control: the inside must not move the fence

**Story:** an agent inside a sandbox edited the project's sandbox config. The change reached the outside within two seconds — including *where credentials were sent*.

- Anything the inside can **write**, the outside must not **trust**: repo config, compose files, scripts.
- The reverse is just as important — repo files that **run on your computer**:
  - dev container `initializeCommand`
  - editor tasks and recommended extensions
  - git hooks and aliases, `.envrc`
- Fix: the outside runs a copy **you approved**, after reading the diff.

# A repo edit does nothing until you adopt it

Your computer runs the copy of foldyard.toml you accepted. Anything in the box can propose a change — only you can make it real.

**THE VM** can write the checkout

### 1 · Someone edits it

An agent in the box, a package install script, a branch you checked out to review: anything that can write the repo can change foldyard.toml.

**The box can't adopt it**

fy config adopt refuses to run inside the box — the answer comes from your terminal.

**YOUR COMPUTER** runs only what you adopted

### 2 · Noticed, not applied

The supervisor keeps running the adopted copy. It logs the change once, fy doctor shows a warning, and on macOS you get a notification.

### 3 · Asked before launch

fy up, fy box up, fy host restart and fy code show you the diff and ask before starting anything.

### 4 · Adopted

Takes effect within a couple of seconds, no restart. The copy lives in ~/.foldyard/adopted/, outside the repo.

Ignore keeps the adopted copy and asks again next time. Revert puts the file back.

```
⚠ foldyard.toml changed since the host adopted it [main]
  +passthrough = ["@all"]
  [a]dopt · [r]evert the file · [i]gnore for now (default):
```

**THE EDGES** where it's strictest

### The first time

A fresh checkout shows its config and asks: adopt, or quit. Until then your computer runs nothing for it: no proxy, no token services.

### No terminal

CI or a detached launch never adopts. A change keeps the old copy running; a checkout with nothing adopted refuses to start and prints fy config adopt.

### A new worktree

Its first adoption is compared with main's adopted copy — never main's working tree — so you see only what this branch would add.

### The commands

```
fy config status
fy config diff
fy config adopt
fy config revert
```

on your computer only

## What it covers

**Adopted**

Everything your computer does with it: the token services, where credentials are injected, passthrough, the switches fy mode shows, [machine].

**Read from the checkout**

Compose files, [box] tools and [ports]: they shape the VM, which untrusted code can already reach. (The compose client gets a minimal environment.)

**Before you say yes**

fy config widenings lists what the config lets through: hosts left undecrypted, where each credential goes, which prompts steer the agent.

**The general rule**

Code inside the fence must not be able to move the fence. The file is inside; the decision to act on it is not.

## Verify, don't trust

- A sandbox should **test itself from the inside**: what's mounted, which escapes fail, which credentials are absent, whether traffic that skips the gate is refused.
- **Test the test.** One self-check printed *ALL PASS* when its probes had never run; the fix was a control that must succeed first, proving each probe ran.
- Re-run it after every config change.

> *If the only evidence that the fence holds is the vendor's README, then you don't know.*

## What no sandbox fixes

- **It can still change your code.** Review diffs; let CI and branch protection do their job.
- **Access you switch on can be used while it's on.** Don't switch on a credential and then run an untrusted install.
- **A dirty repo stays dirty.** Secrets in the project folder are inside the fence.
- **Full autonomy inside is still full autonomy.** An agent can wreck its own sandbox — that's the point of making it throwaway.
- **Humans approve things.** A diff nobody reads is a gate nobody guards.

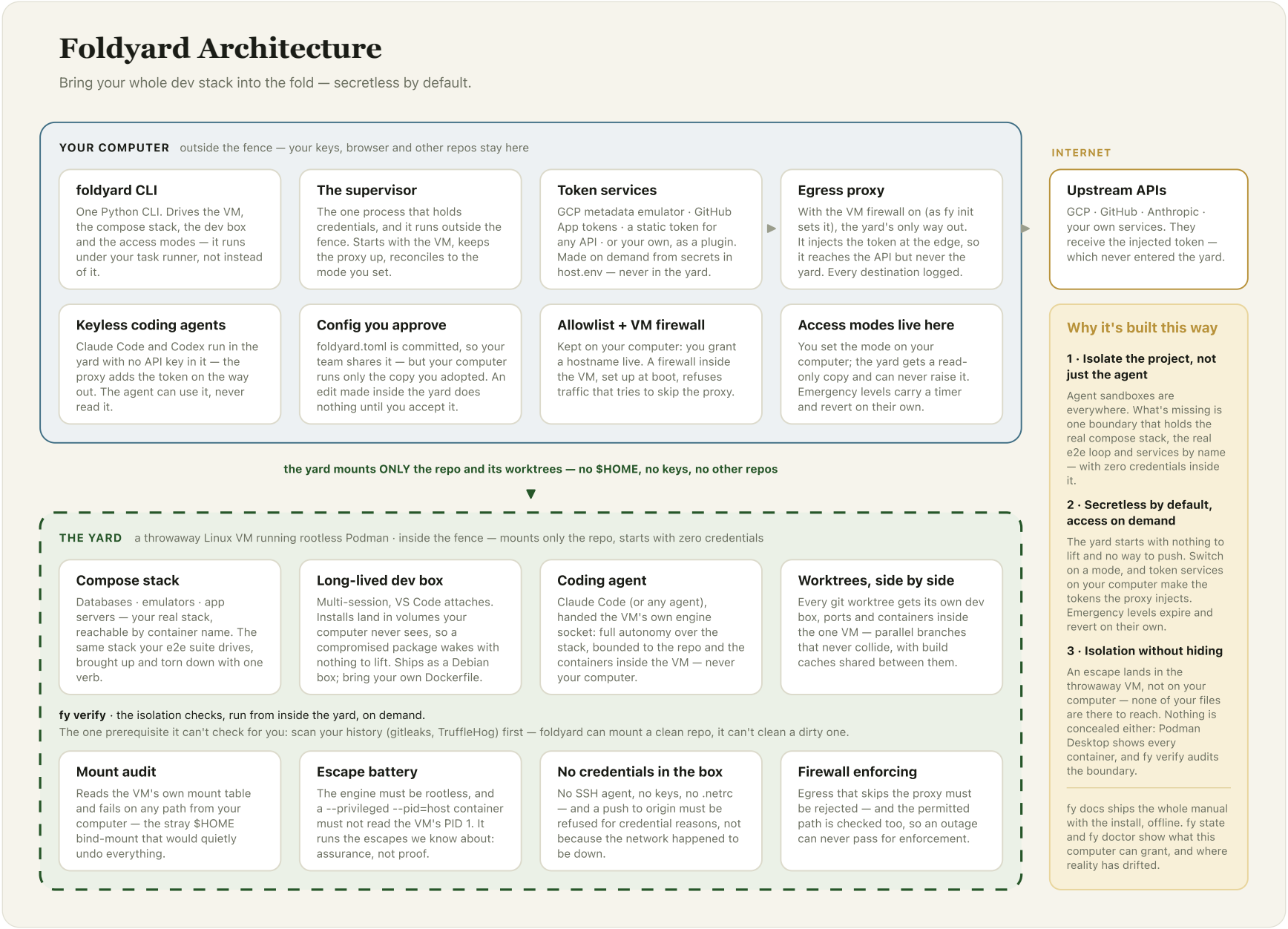
## Questions to ask any sandbox

1. Does it have **its own kernel** (a VM), or share yours?
2. What exactly is **mounted** — and is your home folder anywhere near it?
3. Where do **credentials** live? Can the inside ever read one?
4. Is egress **enforced**, or only routed when software cooperates?
5. Can the inside **edit the config** that the outside obeys?
6. Does it ever **run repo files on your computer**?
7. Does it hold your **whole dev loop** (database, services, tests, editor), or just one agent run?
8. Can you **verify** it yourself?

PART 3

# One answer: foldyard

# The one-glance picture



# How foldyard answers the questions

| Question | Foldyard's answer |
| --- | --- |
| Own kernel? | Yes — a throwaway Linux VM per project (Lima), rootless Podman inside |
| What's mounted? | The repo and its worktrees. No home folder |
| Credentials? | None by default. Switched on per job, mostly swapped in by the proxy; "act as me" levels expire |
| Egress? | Every request goes through a proxy on your computer; with `[machine] firewall = true`, the VM firewall refuses anything else |
| Inside edits config? | Inert until you read the diff and adopt it |
| Repo code on your computer? | foldyard runs none: its config is data. Your git still runs hooks |
| Whole dev loop? | Yes — your compose stack, the dev box, your editor's backend, the agent |
| Verify? | `fy verify`: mounts, escape attempts, credentials, traffic that skips the proxy refused |

# Harden it one step at a time

## Foldyard Isolation Layers

The same VM, hardened one ring at a time — each step is one line in foldyard.toml, and the credentials never move.

**THE CONTAINMENT STACK** what an escape from the box has to get through, inside → out · cumulative · dashed = a boundary, thick = one that also refuses egress

### 1 · Podman VM

`[machine] backend = "podman"`

MACOS · LINUX

**HOST** keys · supervisor · proxy

**MACHINE VM**
one machine, shared by every project
rootless podman, one engine socket
mounts only repo + worktrees root
no firewall — egress is the dial below

**dev box**
compose stack · the agent and everything they create

engine socket ▶ the box

The floor, and already a real fence: containers need a Linux kernel, so the VM was coming anyway — nothing to install beyond podman. An escape lands in a throwaway VM, not your computer. What it lacks: a VM per project and a firewall — traffic can skip the proxy.

### 2 · Lima + VM firewall

`[machine] firewall = true`

MACOS · LINUX

**HOST** keys · supervisor · proxy

**MACHINE VM**
one VM per project, side by side
hypervisor pinned: vz or qemu
mounts only repo + worktrees root
firewall: nftables, installed at boot

**dev box**
compose stack · the agent and everything they create

engine socket ▶ the box

Today's default (lima is the default backend). The hypervisor is pinned, and root in the guest is boot-time only: the firewall goes in at boot, out of the box's reach. Traffic that ignores the proxy is refused, not let through; verify checks both directions.

### 3 · + gVisor EXPERIMENTAL

`[machine] runtime = "gvisor"`

MACOS (M1/M2 TOO) · LINUX

**HOST** keys · supervisor · proxy

**MACHINE VM**
as step 2, plus a second API socket
runsc by default, pinned + verified
wrapper fixes the flags, no override

**GVISOR** the box's own kernel

**dev box**
compose stack · the agent and everything they create

narrowed socket ▶ the box

The box, and everything it creates, runs on gVisor's userspace kernel: an exploit must beat the Sentry before it reaches the kernel with the socket and mounts. A VM setting, not a box knob: the socket the box holds strips the runtime opt-out, so the box cannot leave it.

### 4 · + host firewall

`[machine] host_firewall = true`

LINUX ONLY

**HOST** nftables on the VM's own slice

**MACHINE VM**
as step 3
the VM runs in its own systemd slice
so your computer knows its packets

**GVISOR** the box's own kernel

**dev box**
compose stack · the agent and everything they create

narrowed socket ▶ the box

The firewall again, out of the guest's reach: your computer matches the VM's slice and allows only the proxy ports. A guest-kernel exploit that flushes the VM firewall still meets this one. You install it once; every fy up checks it. macOS can't: the VM's net runs as you.

**THE WAY OUT** egress from the VM — one dial, four positions; grants of every level live on your computer, never in the repo

the box ▶ VM firewall ▶ host firewall ▶ **egress proxy — the token is injected here** ▶ upstream API

### Open

`[proxy] absent`

A clean box: no proxy env, no CA mounted. Egress is direct, unlogged, and no firewall is possible — it needs the proxy as the only way out. For a stack that never talks out, or one you trust entirely.

### Observe

`[proxy] · enforce = false`

Everything goes through the proxy on your computer: decrypted and logged, passthrough hosts as-is. Nothing is blocked; the log shows what the box reaches for. The token is already injected here.

### Enforce

`enforce = true · "learn"`

A host that isn't granted gets 403 and is never dialled. "learn", which init writes, only records for an hour, then enforces by itself. Grants live on your computer, never in the repo, which the box can write.

### Fail-closed

`firewall · host_firewall`

Steps 2–4 above: traffic that ignores the proxy env is refused in the VM, and on Linux on your computer too. verify checks the refusal AND the permitted path, so an outage never reads as enforced.

### What never moves

**1 · The credentials**

Whichever ring you pick, the token is injected at the proxy edge, on your computer. A VM-root escape can flush a firewall; it steals no credential.

**2 · The socket is the design hole**

The box drives the stack through the engine's own API, and an API can mount whatever its uid can read. Under gVisor it is narrowed to strip the runtime opt-out; a mount allowlist is deferred — the VM holds one project and no key.

**3 · fy verify proves the ring you're in**

PID 1's mount table, the escape battery, the firewall in both directions — and under gVisor, the kernel the box actually runs on.

Cost, measured on macOS: gVisor is ~1.2× on a test suite and 2–4× on sub-second git and lint calls; the firewalls cost nothing at runtime.

All of it, in foldyard.toml:

```
[machine]
backend   = "lima"
firewall  = true
runtime   = "gvisor"
# Linux only:
host_firewall = true
[proxy]
enforce = true
```

## If foldyard isn't for you

It was made with a startup's needs in mind — pick by your threat model, not by ours.

- **Just one agent, short tasks?** A per-task agent sandbox (VM- or process-based) may be all you need.
- **Happy with closed source and a Docker account?** Docker Sandboxes also fits a whole stack in a VM, with credentials kept outside.
- **Already on dev containers?** Harden them: rootless engine, no home-folder mount, no secrets in env, an egress gate.
- **Can't run a VM locally?** A remote dev environment moves the whole risk off your laptop.
- **Whatever you pick:** run it through the eight questions.

Foldyard's own survey of the landscape: `docs/prior-art.md` in the foldyard repo.

PART 4

# Hands on

# From nothing to a walled box

## From nothing to a walled box

Do the big downloads once, before the day. Then each project's VM is a clone, and its box is seconds away.

**ONCE PER COMPUTER** at home, before the workshop — the part that needs a good connection

**1 Install uv**
`docs.astral.sh/uv`
The one thing you install by hand. On Windows: WSL2 first, then everything inside its Linux.

**2 Check this computer**
`uvx foldyard@latest prereq`
Checks Lima, podman, a hypervisor (on Linux: QEMU, KVM, systemd), disk and memory. Prints the install command for anything missing.

**3 Install foldyard**
`uv tool install foldyard`
Puts foldyard on your PATH, with fy as shorthand. mitmproxy, which runs the proxy, comes with it.

**4 Bake the base VM**
`fy prereq --fetch`
Downloads the VM image, installs podman, builds the box image — once, about two minutes. Nothing mounted, no repo code.

**on the day: seconds, and almost no downloads**

**PER PROJECT** on the day, in your own repo

**5 Write the config**
`foldyard init`
A locked-down foldyard.toml: Lima, the VM firewall on, the allowlist learning for its first hour. Agents and your stack are commented out.

**6 Start the box**
`fy box up`
Your project's VM, cloned from the base. Shows you the config to adopt and the hosts the repo recommends — one yes each.

**7 Step inside**
`fy box shell`
A shell in the box: the repo, your tools, no credentials. Commits work; pushes are refused.

**8 Check, don't trust**
`fy verify`
Mount audit, the known escapes, credential checks, firewall probes. It should say ALL PASS.

**9 Put an agent in**
Uncomment [claude] or [codex], fy config adopt, fy box down && fy box up, fy mode claude=on, fy claude. The key stays outside.

**10 Bring your stack in**
Uncomment the compose lines in [project], then fy up: your whole stack runs in the same VM, beside the box.

**WHAT FY BOX UP DID** step 6, opened up, in the order it happens

**Your config, adopted** — the first time, you read it and press Enter to accept

**A VM for this project** — a clone of the base, then its own mounts and firewall

**The supervisor** — starts the proxy on your computer, waits until it's ready

**The box image** — re-tagged from the base, not rebuilt

**The dev box** — started with the proxy's CA and settings — and no keys

### The numbers

A first fy box up, without and with a base:

| | without | with |
|---|---|---|
| download | ~500 MB | none |
| first boot | ~30 s | ~10 s |
| podman | installed | there |
| box image | built | reused |
| **fy box up** | **minutes** | **~26 s** |

Measured once on an Apple-silicon macOS computer (Lima 2.1.3, vz); the bake took 105 s.

### What keeps it safe

**1 · Bake on your own computer**
A VM disk from someone else's computer is one you didn't see being made. Share Lima's and uv's download caches instead.

**2 · Never made from a project**
The base is built with nothing mounted and no agent ever run, so nothing escapes into it.

**3 · Cloned only at creation**
An existing project VM is never replaced. fy machine base says when the base is out of date.

Left for the day: your project's own images, packages and agents — the part no base can fetch ahead of time.

# Today's plan

| | Do | Look for |
|---|---|---|
| 0 | `fy prereq` | no ✕, base VM ✓ |
| 1 | `cd your-repo` → `foldyard init --with vscode` | a `foldyard.toml` to read |
| 2 | `fy box up` → `fy box shell` | "cloning the base VM" — seconds, not minutes |
| 3 | `fy verify` (on your computer, then in the box) | ALL PASS |
| 4 | `fy code` → uncomment `[claude]` or `[codex]` in the box → `fy config diff` on your computer → `fy config adopt` → `fy box down && fy box up` → `fy code` → `fy mode claude=on` → `fy claude` | the box's edit in the diff, then a hidden prompt for your key, once |
| 5 | ask the agent to fill in your stack → `fy config diff` → `fy config adopt` → `fy up` → `fy open` | your app, running in the VM |

The allowlist **learns for its first hour**. When it starts refusing, `fy allow learn` grants what it saw, after you've reviewed it.

## When something's off

- `fy doctor` diagnoses; `fy tui` shows the same checks, with one-click fixes.
- **Changed the config and nothing happened?** It needs adopting: `fy config diff`, then `fy config adopt`.
- **Agent gets 401s?** The switch is off: `fy mode claude=on`.
- **A request refused?** Find it in `fy tui`'s Network Log and grant it there, or `fy allow add <host>`.
- **Everything's slow or swapping?** Lower `memory_mib` under `[machine]`, `fy config adopt`, then `fy machine recreate`.
- **Reading more:** `fy docs` — the whole manual, offline, matched to your version.

## Sources and further reading

- Simon Willison, The lethal trifecta for AI agents (June 2025)
- Shai-Hulud: Krebs on Security (Sep 2025); Tenable — Mini Shai-Hulud FAQ (2026)
- Nx "s1ngularity": Wiz (Aug 2025)
- TrapDoor: The Hacker News (May 2026)
- VS Code extension breach: BleepingComputer; CSA research note (May 2026)
- LiteLLM 1.107.0: LiteLLM's attack chain write-up; Datadog Security Labs (Oct 2026)
- Foldyard: github.com/twistco/foldyard — `docs/archive/lessons.md` has more stories from building it

# Questions?