dotfiles bootstrap

Key takeaways

1. Split setup scripts by what they mutate, not by which platform runs them.
2. A symlink stores a path, and paths can dangle — verify the artifact, not the log.
3. Agent configuration — instructions, skills, hooks — is dotfiles too; version it.

Common misunderstandings of portable bootstrap scripts

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

"Separate setup scripts by operating system to isolate platform logic."

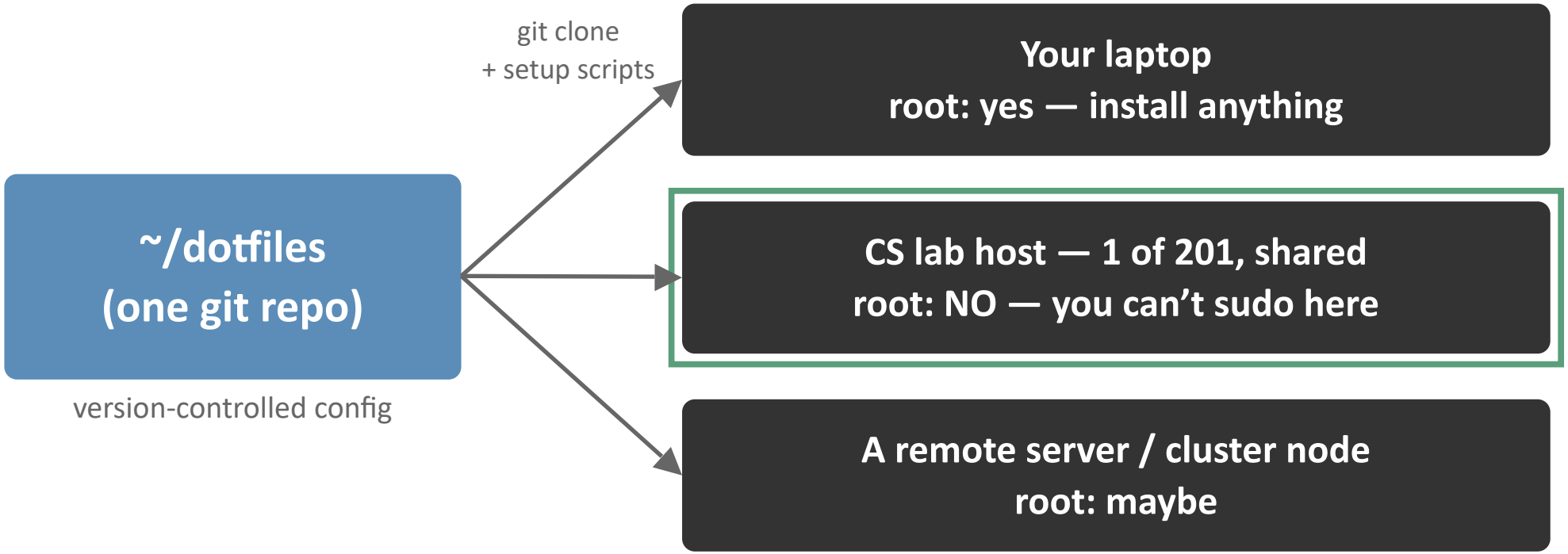
"Instruction files are documentation; only code can change a program's behavior."

"Agent configuration is per-machine tuning that has no place in version control."

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

Your environment follows you — even without root

One config, every machine — including the ones you can’t sudo on



Everything except system-package installs needs no root — so your shell, editor, aliases, and agents follow you even onto a shared machine you can’t install software on.

One repo carries your environment to every machine

HW0's ./setup is a bootstrap — today is what that kind of script does, and why

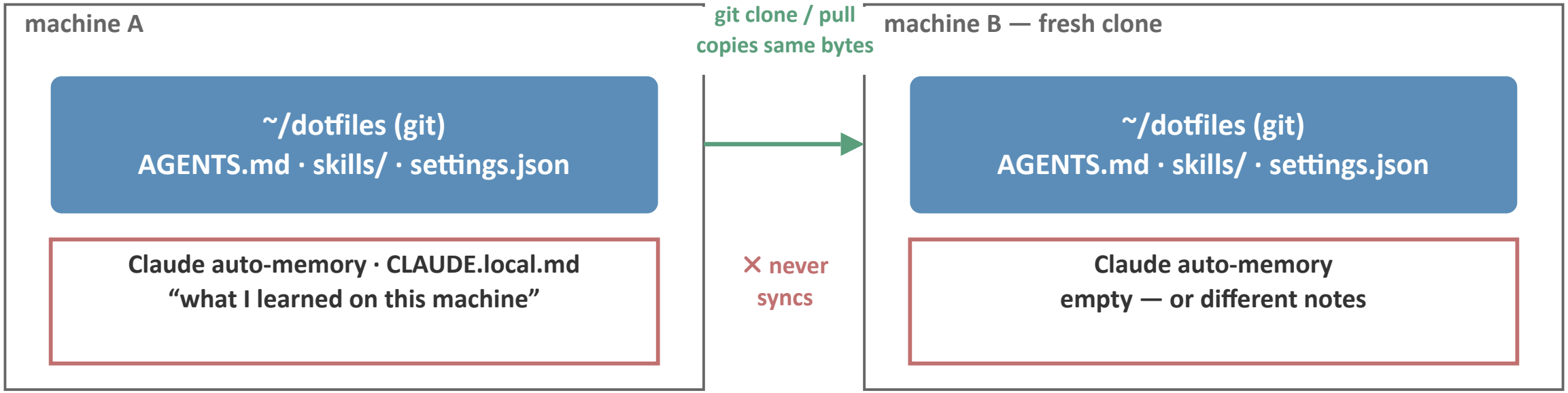
One script per kind of mutation, not one script per OS

```
$ gh repo clone utac-f26/dotfiles-<login> ~/dotfiles  # your private repo (012)
$ cd ~/dotfiles
$ ./packages   # system packages via apt/brew/pacman — the only sudo step
$ ./links      # symlink config (.bashrc, .gitconfig, ...) into $HOME — as you
$ ./tools      # user-space CLIs (uv, node, gh, rg) into ~/.local — as you
```

- Dotfiles: the config files in your home directory, checked into one git repo.
- Bootstrap: the steps that turn a fresh clone into a working environment. Each machine runs them once — but you keep meeting new machines, so the steps live in scripts, not in your memory.
- One design rule runs through everything today: route setup by what gets mutated, not by which OS is running.

No local memory files — all state is explicit

If it matters, it is in the repo; nothing important lives anywhere else



Memory files are state the agent accumulates behind your back — unversioned, unreviewed, different on every machine.

Course policy, in your seeded AGENTS.md: durable knowledge lives in the repo — never in memory files.

New account on a new machine? Check out ~/dotfiles, run ./setup, and you are in an agent environment that is known and comfortable.

Instapoll

Instapoll · Review · Aug 27 B

Which bootstrap property would save you the most time after replacing a laptop?

- A. It recreates the required tools and configuration from version-controlled setup files
- B. It preserves every cache and temporary file from the old machine
- C. It installs optional applications before verifying core dependencies
- D. It assumes the new operating system already matches the old one

Which axis should organize a bootstrap?

Several designs are defensible — this repo lived in the middle one before choosing the third

One script per OS

setup-macos, setup-linux, setup-windows. The steps are the same everywhere, so every shared fix must land three times.

Duplicates exactly the part that is shared.

One script, many flags

This repo's past: symlinks plus package installs in one ~900-line script, a flag per platform and job (--tools, --pacman, …).

'Which lines need sudo?' has no answer.

One script per mutation

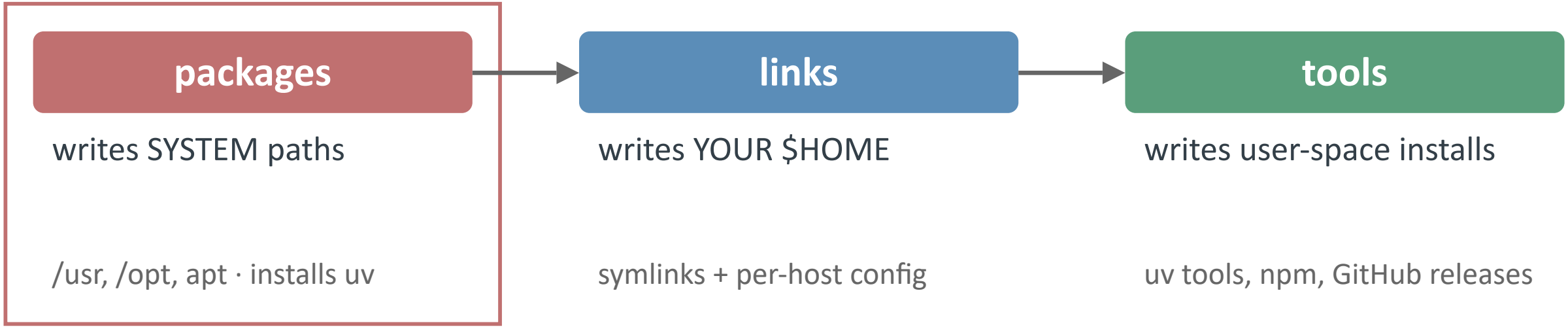
Route by what gets written: system paths, your $HOME, user-space installs. The OS shrinks to a small branch in each script.

The split we chose — next slide.

After the split: one script per kind of write

Route by what gets mutated, not by which OS

needs sudo (Linux) / UAC (Windows)



recommended run order: packages → links → tools

during the semester: ./packages --update and ./tools --update refresh what is already installed

Only packages ever needs elevation. links and tools run as you — so ‘what runs as root?’ has a one-script answer.

Anatomy of ~/dotfiles/claude/

Your agent's configuration is dotfiles too

```
~/dotfiles/claude/
  AGENTS.md                  # shared agent instructions
  CLAUDE.md                  # @AGENTS.md shim for Claude Code
  settings.json              # permissions, env vars, hooks
  agents/                    # named Claude subagents
  commands/                  # slash commands
  skills/                    # reusable workflows
  bashenv.sh                 # non-interactive shell setup
  breadcrumbs/               # /clear digests (runtime, gitignored)
  handoffs/                  # saved handoff notes (runtime, gitignored)
```

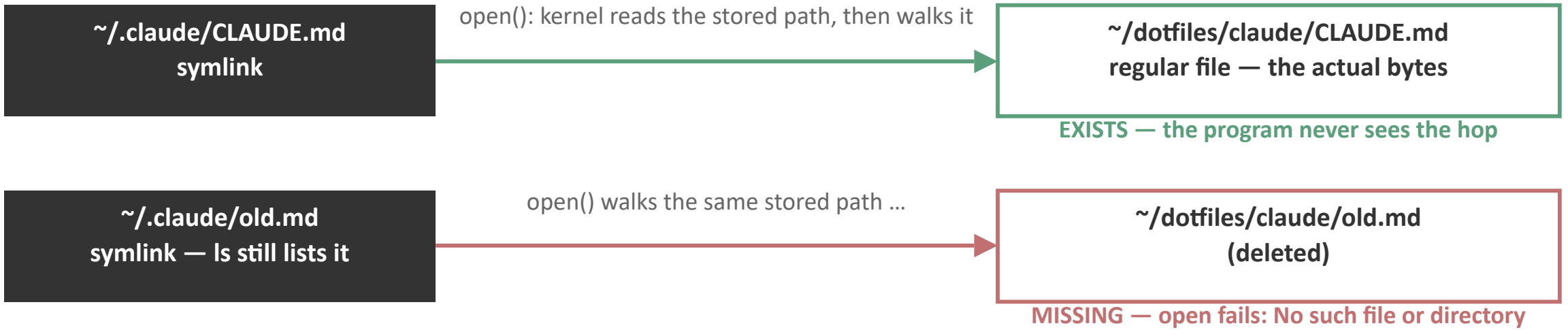
- One AGENTS.md feeds every agent. Claude Code reads CLAUDE.md, Codex reads AGENTS.md — so CLAUDE.md is a one-line @AGENTS.md shim.
- Claude actually reads ~/.claude/, not this directory — ./links bridges that gap with symlinks. Next slide: what a symlink is.

A symlink is a file that stores a path

The kernel re-walks that stored path on every open — and nothing stops it from dangling

The size is 36 bytes: the link's entire content is that 36-character path

```
$ ls -l ~/.claude/CLAUDE.md
lrwx------ 1 you 36 Aug 25 ~/.claude/CLAUDE.md -> /Users/you/dotfiles/claude/CLAUDE.md
```



./links creates all of these links and is idempotent — rerunning it is always safe. Edit either path; the one real file lives in the repo.

What is a skill?

A named procedure the agent loads only when the task matches

- Definition
  - A skill is a reusable workflow: a directory holding a SKILL.md procedure plus any helper files.
  - The agent sees only each skill's one-line description; the full procedure loads when a task matches it.
- Think: a macro for agent work
  - Like an editor macro, a skill replays the same sequence of actions on demand: code-maintenance runs the same staged pass on any codebase — baseline the tests, inventory duplicates, tighten types, hunt defects, verify — same stages, same order, every time. Next slide walks through it.
  - Unlike a macro, the steps are prose the agent interprets — it can check preconditions and stop when one fails instead of replaying blindly.
- Why not just more AGENTS.md prose?
  - AGENTS.md is read every session, so every line costs context on every task.
  - A skill's ~180 lines cost nothing until the matching task shows up.
- Where it lives
  - ~/dotfiles/claude/skills/<name>/SKILL.md — tracked, reviewed, and versioned like any dotfile.

A skill in practice: code-maintenance

Find the bugs and logic errors, add the missing tests, delete the stale ones… — tedious to type every time, so it became a skill

```
skills/code-maintenance/SKILL.md      # ~180 lines of staged procedure
description: maintenance pass, cleanup, dead-code removal, consolidation,
  stronger typing, defect hunt.  Not feature work.        # the trigger
```

How to do a maintenance pass	The discipline each stage encodes
establish the contract	find the validation commands and baseline failures first
inventory before editing	trace duplicates through callers; one canonical mechanism survives
improve types and APIs	narrow exports; make illegal states hard to represent
hunt for defects	probe empty, boundary, and cleanup paths; each real bug gains a test
preserve security and trust	isolation, compatibility, and durability may not weaken under ‘cleanup’
verify, then report	lint, typecheck, full suite, whole-diff read; report net line counts

Instruction files are part of the program

A real bug fix: seven instruction files changed, the behavior changed — runtime code did not

Before — broad prompt invites busywork

```
Rename variables to express intent
Replace literals with constants
Fix direct file overwrites
Narrow broad exception handlers
Ensure proper cleanup

# agent hears: touch everything
```

After — instructions encode judgment

```
Rename only if truly ambiguous
Short names OK in small scopes
Atomic writes for key state only
Broad catches can be defensive
Close only what actually leaks

# agent hears: change less
```

Files that change behavior deserve what code gets — review, history, version control. That is why claude/ lives in your dotfiles repo.

What is a hook?

A command the agent runtime runs for you at fixed events

- Definition
  - A hook is a shell command registered in settings.json that the agent runtime runs at a lifecycle event.
  - Events include session start and end, before and after each tool call, and prompt submission.
- Why a hook instead of an instruction?
  - A hook fires deterministically every time; an instruction depends on the model applying it.
  - Bookkeeping and guardrails belong in hooks; judgment calls belong in AGENTS.md.
- Where it lives
  - The wiring is a few lines of settings.json; the script it runs is a tracked file in ~/dotfiles.

A hook in practice: format on every edit

The first recipe in Anthropic’s hooks guide — no file the agent touches stays unformatted

claude/settings.json — after every Edit or Write, pull the file’s path from the event JSON and format it

```
"PostToolUse": [{"matcher": "Edit|Write",
    "hooks": [{"type": "command",
        "command": "jq -r '.tool_input.file_path' | xargs npx prettier --write"}]}]
```

- Every diff you review is already formatted — the hook ran before you ever saw the file. For Python, the same line runs ruff format instead.
- An instruction (‘always run the formatter after editing’) depends on the model remembering; the hook fires every time, mechanically.
- The course dotfiles wire richer hooks to other events — on /clear they write a breadcrumb digest so restarts are cheap. Deck 020 (token efficiency) teaches that workflow.

settings.json is a tracked dotfile: clone the repo and every machine gets the same hooks.

What not to put in dotfiles

Portable configuration is not runtime state

No secrets

OAuth tokens, API keys, cookies, private keys, and decrypted env files stay out of git. One push publishes them permanently.

Keep encrypted env files in ~/dotfiles/secrets/.

No transcripts

Session histories are output the agent produced, not configuration you chose. On any other machine they are noise.

Run git status before every dotfiles commit.

No caches

uv, npm, model, and tool caches are per-machine state. They are large, they go stale, and a command rebuilds them.

Track the install command, not its output.

The repo stores how to recreate the environment, not the private state produced inside it.

Key Takeaways

1. Many bootstrap designs are defensible. Split scripts by WHAT they mutate and ‘what runs as root?’ has a one-script answer on every platform.
2. A symlink stores a path the kernel re-walks on every open — so a link can dangle, and exit code 0 is not evidence. Verify the artifact, not the log.
3. The same repo carries your agent: AGENTS.md, skills, and hooks are versioned config; secrets, transcripts, and caches stay out.