INTERACTIVE · OPEN-SOURCE · FREE

The Missing Package Manager

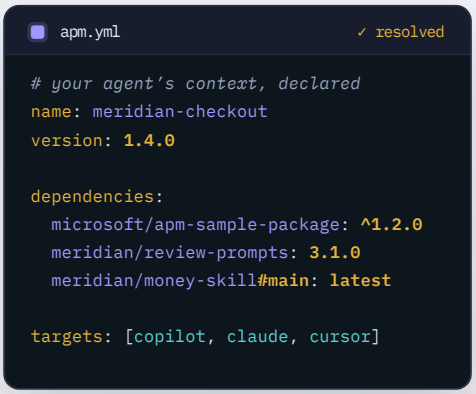
Managing AI Agent Context with APM

Portable · **Reproducible** · **Secure** · **Governed**

12 chapters · verified against the real apm CLI

Maxim Salnikov · Microsoft

apm.isainative.dev



The Missing Package Manager — Managing AI Agent Context with APM

By Maxim Salnikov · Read online at apm.isainative.dev

Edition v1.1 · Updated July 8, 2026

Free interactive edition · Full-book PDF · Generated July 09, 2026

Portability · Reproducibility · Security · Governance

Contents

12 chapters, 6 parts. Concept before command — every APM feature is introduced as the implementation of one of four properties.

PART I · WHY CONTEXT NEEDS A MANIFEST / CHAPTER 1

The Context Problem

Articulate why agent context needs a package manager and name the four properties APM is designed to protect.

Objective

After this chapter you can explain why the files that configure an AI coding agent behave like a project dependency, describe the drift that appears when they go unmanaged, and name the four properties a package-manager frame protects — PORTABILITY, REPRODUCIBILITY, PROVENANCE / SECURITY, and GOVERNANCE — and how APM's three promises deliver them.

Concept/Theory

Agent context is a dependency you never declared

Before npm, pip, and Cargo, application dependencies lived wherever each developer happened to put them: a `lib/` folder copied between machines, a zip emailed around, a README that listed "install these seven things first." It mostly worked — until it didn't. Versions drifted, "works on my machine" became a punchline, and nobody could say with confidence what was actually running or where it came from. Package managers fixed this by turning scattered, hand-copied files into a *declared* dependency: one manifest that names what you need, one lockfile that pins exactly what you got, and one command that restores it anywhere.

The files that configure an AI coding agent are in that same pre-package-manager state today. Your *agent context* — the instructions and coding standards, the reusable prompts, the skills and agent personas, the plugins, and the Model Context Protocol (MCP) server configurations — quietly shapes the code an agent writes, how it reviews changes, and which external tools it can reach. That makes it a real project dependency. Yet most teams still hand-maintain it as loose files scattered across per-harness directories and personal home folders, with no single declared source of truth. APM's own framing is blunt: agents "need context to be useful … but today every developer sets this up manually. Nothing is portable nor reproducible. There's no manifest for it" (microsoft/apm README).

The drift problem

Undeclared, per-machine context drifts. Two developers believe they share a setup, but their instructions, prompts, or MCP servers have quietly diverged — the classic "*works on my machine*" failure, now aimed at AI agents. A shared README doesn't save you: documentation describes *intent* but never restores *state*, so drift returns the moment someone edits a local file. That is the difference APM draws between "follow this 12-step README" and a single install command (the three promises).

• MERIDIAN

You are a staff engineer at **Meridian**, a fintech running a hosted checkout service. Six developers share the `meridian-checkout` repo across three harnesses — GitHub Copilot in pull requests, Claude Code for deep refactors, Cursor for the dashboard. Preparing for a security review, you go looking for "the" agent setup and instead find the drift:

- `.github/copilot-instructions.md` tells Copilot to use the `Money` value object, while Claude's local `CLAUDE.md` still mentions raw decimals.
- Three copies of `review.prompt.md` live in three home directories and one branch — each checking a *different* threat model.
- Cursor rules emphasize React accessibility; Copilot rules emphasize transaction idempotency. Neither is wrong, but neither is portable.
- Onboarding Anika, the new joiner, means a wiki page, a Slack thread, and copying prompt files from a colleague.
- Security cannot answer a simple question: *which MCP servers are installed, and did each prompt come from an approved source?*

None of this is declared anywhere. It is dependency surface — it just was never treated like one.

The four properties a package-manager frame protects

Reframing agent context as a dependency exposes four properties that go missing when it drifts. This book uses them as a lens in every chapter, so keep them handy:

- PORTABILITY — the same declared context can be materialized on any machine and any supported harness. Absent it, every developer's agent behaves a little differently and onboarding is manual.
- REPRODUCIBILITY — a later restore reproduces the previously known-good context *exactly*, down to the bytes. Absent it, a moving branch or an upstream edit silently changes what a "review prompt" actually contains.
- PROVENANCE / SECURITY — you can tell where each piece of context came from, and risky content is checked before it reaches disk. A prompt is, in effect, a program for an LLM, so an unvetted prompt or transitive MCP server is real supply-chain surface (the three promises).
- GOVERNANCE — an organization can decide which sources, scopes, and primitives are allowed, and have that decision enforced automatically. Absent it, policy lives in people's heads and is applied inconsistently.

Two distinctions are worth holding onto, because the rest of the book leans on them. Portability is about *where* context can land (any machine, any harness); reproducibility is about whether it lands as the *same* bytes. And security asks "is this safe?" while governance asks "is this allowed here?" The next section maps all four onto what APM actually ships.

In APM

Three files and one habit

APM borrows the manifest-plus-lockfile shape from npm, pip, and Cargo and applies it to the files that configure AI coding agents (What is APM?). Conceptually it is **three files and one habit** — you don't need any syntax yet, just the map:

- `apm.yml` — the **manifest.** The single human-authored source of truth: it names the primitives (skills, prompts, instructions, agents, plugins) and MCP servers your project depends on, and which harnesses to target. This is what carries PORTABILITY.

- `apm.lock.yaml` — the **lockfile**. Machine-generated, never hand-edited: it pins every resolved dependency to an exact source ref *and* a content hash, so two developers who install against the same lockfile get byte-identical context. This carries REPRODUCIBILITY and half of PROVENANCE.
- `apm-policy.yml` — **install-time governance**. An organization's allow/deny rules, enforced when context is installed (including transitive MCP servers). Unlike the two files above, it isn't a per-repo default: it's an *org-level* artifact APM discovers from your organization's git remote (a `.github`, `.apm`, or `_apm` location), authored once in Chapter 9 rather than kept in every developer's checkout. This carries GOVERNANCE.
- `apm install` — the **habit**. The one command that reads the manifest (and lockfile) and restores the declared context. Onboarding stops being a 12-step README and becomes `git clone` then `apm install` (the three promises).

You author the manifest first; APM generates the lockfile on install, and organization policy lives outside the repo entirely. Chapter 1 only names these files — layout and exact syntax arrive from Chapter 4 onward.

Four properties, three promises

The four properties are *this book's* teaching frame. APM itself commits to **three promises**, described in the docs as "deliberately small and load-bearing": portable by manifest, secure by default, governed by policy. Here is the bridge, so no property is left an orphan slogan:

Property (book frame)	APM promise (official)	What carries it
PORTABILITY	Portable by manifest	`apm.yml` + harness targets
REPRODUCIBILITY	(carried under "portable by manifest")	`apm.lock.yaml` — exact refs + content hashes
PROVENANCE / SECURITY	Secure by default	content hashes + install-time scanning
GOVERNANCE	Governed by policy	`apm-policy.yml`

How the book's four properties map onto APM's three official promises.

The book deliberately splits *reproducibility* out as its own property even though APM files it under the first promise. That is a teaching choice, not a fourth promise in the docs — a distinction the pitfalls below make sharper.

One context, many harnesses

A *harness* is a specific AI coding tool that reads agent-context files. **Harness parity** is APM's guarantee that one declared context restores the same skills, prompts, instructions, MCP servers, and more across all of them: "any developer runs `apm install` and gets the same agent context across GitHub Copilot, Claude Code, Cursor, OpenCode, Codex, Gemini, Windsurf, and Kiro" (APM docs). APM compiles one source into each tool's native location — `.github/` for Copilot, `.claude/` for Claude Code, `.cursor/` for Cursor — so the same intent lands in the harnesses you declare or APM detects, not fanned out to every tool by default (target resolution is covered in Chapter 5). Meridian's three-harness mix stops being three maintenance burdens and becomes one.

Crucially, APM writes the files each tool already understands and then "stays out of the way at agent runtime" (What is APM?). It restores context; it does not run, proxy, or sandbox the agent.

◆ FOR ENGINEERING LEADERS

Notice what moves onto the balance sheet. Once agent context is a declared, reviewed file, three costs shrink: **onboarding** (a clone-and-install replaces the wiki-plus-Slack ritual), **audit** (you can answer "which tools and prompts are installed, and from where?" from a lockfile instead of a survey), and **supply-chain risk** (organization policy is enforced automatically at install time rather than trusted to individual judgment). The reframe is simple: agent context deserves the same declaration and review discipline you already require of `package.json` or `requirements.txt`.

When to use / pitfalls

When a package-manager frame helps

APM earns its keep when context is *shared* and *plural*:

- **Multiple developers** depend on the same instructions, prompts, or skills — and drift between them costs real time.
- **Multiple harnesses** are in play (Meridian's Copilot + Claude Code + Cursor), so hand-maintaining per-tool files guarantees divergence.
- **Onboarding** should be reproducible, not a scavenger hunt across home directories.
- **Security or compliance** needs to inventory what agent tooling is installed and where it came from.
- **An organization** wants to enforce which sources and MCP servers are allowed.

When it's overkill

Be honest about the cases where a manifest is ceremony you don't need:

- A **solo developer** on a **single harness** with a couple of stable instruction files — the drift surface is tiny.
- **Throwaway or spike** work you won't share or restore.
- Context that is genuinely a **personal preference** (your own favorite prompts), not a project dependency the team relies on.

The dividing line is the one package managers drew decades ago: the moment context is shared, restored, or audited, "keep copies in sync by hand" stops scaling.

⚠ **Pitfall: four properties are the frame, not APM's promises**

Don't tell a colleague "APM makes four promises." It doesn't. The docs commit to exactly **three**: portable by manifest, secure by default, governed by policy (the three promises). *Portability, reproducibility, provenance/security, and governance* are this book's teaching lens; reproducibility lives *inside* the first promise. Use the four to reason; cite the three when you quote APM.

⚠ **Pitfall: policy governs installs, the harness governs runtime**

"Governed by policy" does *not* mean APM controls what your agent may do while it runs. As the docs put it plainly: "apm-policy.yml governs what gets installed; your agent harness governs what runs. The two planes do not overlap" (microsoft/apm README). APM is an install-time integrity and governance layer, **not** a runtime sandbox. Expecting it to police live agent behavior will lead you astray.

> **ⓘHeads-up for Chapter 2: names can collide**
>
> Once context comes from many sources, the same primitive *name* can mean different things — exactly the ambiguity that namespaces and lockfiles were built to tame. Chapter 2 draws the lessons package managers already learned (naming, pinning, resolution) and shows which ones APM inherits.

Worked example

Meridian: from scattered files to one declaration

Here is Meridian's drift as an inventory. Nothing below is an APM file yet — it's four hand-edited files, plus an uninventoried set of MCP servers, none aware of the others:

File	Harness	Holds	Problem
`.github/copilot-instructions.md`	Copilot	Backend rules (use `Money`)	Hand-edited; disagrees with Claude
`.claude/CLAUDE.md`	Claude Code	Older rules (raw decimals)	Copied from another repo; never reconciled
`.cursor/rules/frontend.mdc`	Cursor	React accessibility conventions	Frontend-only; not portable to the API work
`~/prompts/review.prompt.md`	(any)	Security review checklist	Three private copies, three threat models
MCP servers	(various)	External tool access	No inventory; security can't say what's installed

Meridian's per-harness context before APM — four sources of truth, none declared.

As a file tree, the anti-pattern is stark — related intent, scattered and unpinned:

Anti-pattern: per-harness context that drifts independently (not an APM layout). `apm v0.23.1`

```
# Anti-pattern only -- four unmanaged sources of truth
.github/copilot-instructions.md       # backend rules, hand-edited
.claude/CLAUDE.md                     # older rules copied from another repo
.cursor/rules/frontend.mdc            # frontend-only conventions
~/prompts/review.prompt.md            # private, unversioned review prompt
```

The fix is not “write a better README.” It's to *declare* the context in one file the repo owns, then let a single command restore it everywhere. Here is the shape Meridian is heading toward — a minimal manifest that names the project and its harness targets, with dependency buckets ready to fill in later chapters:

Conceptual target `apm.yml` for `meridian-checkout` — illustrative only; syntax is validated in Chapter 4. `apm v0.23.1`

```
name: meridian-checkout-agent-context
version: 0.1.0
description: Shared AI-agent context for the Meridian checkout service.
targets:
  - copilot
  - claude
  - cursor
dependencies:
  apm: []
  mcp: []
includes: auto
```

Read it as a promise, not yet a program. `name`, `version`, and `description` identify the package; `targets` lists the three harnesses Meridian uses; `dependencies` splits into two typed buckets — `apm` for primitives and `mcp` for MCP servers — both empty for now; `includes: auto` tells APM to auto-discover locally authored primitives. One reviewable file, three harnesses, one future `apm install`. Chapter 4 turns this sketch into a validated manifest; Chapter 5 runs the install for real.

Recap & next

Recap

- Agent context — instructions, prompts, skills, agents, plugins, MCP configs — is a real project dependency. Unmanaged, it drifts: “works on my machine,” for AI agents.
- A package-manager frame protects four properties: PORTABILITY, REPRODUCIBILITY, PROVENANCE / SECURITY, and GOVERNANCE.
- APM delivers them through three official promises — portable by manifest, secure by default, governed by policy — so the four-property lens maps cleanly onto what APM ships.
- The map is three files and one habit: `apm.yml` (declare), `apm.lock.yaml` (reproduce), `apm-policy.yml` (govern), and `apm install` (restore).
- Harness parity means one declared context restores across many harnesses; APM configures files and stays out of the runtime.
- Meridian's next step: turn four scattered per-harness files into one declared manifest.

Next

Package managers already solved most of this for application code. Chapter 2 — Lessons from Package Managers distills what npm, pip, and Cargo taught us — manifests, lockfiles, pinning, and the name-collision problem flagged above — and shows which lessons APM inherits.

PART I · WHY CONTEXT NEEDS A MANIFEST / CHAPTER 2

Lessons from Package Managers

Map package-manager concepts onto agent context so APM feels familiar rather than novel.

Objective

After this chapter you can map the six things npm, pip, and Cargo normalized — a manifest, a lockfile, version pinning, sources, provenance, and a repeatable restore — onto their agent-context analogues in APM, tell Microsoft's Agent Package Manager apart from the unrelated tools that share the name “apm,” and say honestly where the package-manager analogy stops holding.

Concept/Theory

What package managers actually normalized

Chapter 1 left application dependencies in their pre-package-manager state: copied `lib/` folders, emailed zips, a README that listed “install these seven things first.” npm, pip, and Cargo did not simply automate the download. They *normalized a pattern* — six moves that turned dependencies from hand-copied files into declared, reviewable, reproducible project state. Those six moves are the whole lesson, and every one of them has a direct analogue in agent context:

- **Manifest** — a file that declares what you intend to depend on.
- **Lockfile** — a file that pins the exact resolved graph you actually got.
- **Version pinning** — “some compatible version” becomes “this exact one.”
- **Sources / registries** — a named place a dependency resolves from.
- **Provenance** — knowing where each artifact came from and that it is intact.
- **Repeatable restore** — one command rebuilds the exact same environment anywhere.

APM's own framing is that it “borrows the manifest-plus-lockfile shape from npm, pip, and cargo and applies it to the files that configure AI coding agents” (What is APM?). Here is the one-for-one map — each fundamental, its agent-context analogue, and the Chapter 1 property it protects:

Package-manager fundamental	What it does (npm / pip / Cargo)	Agent-context analogue	Protects
Manifest	Declares intended dependencies as reviewable state (`package.json`, `requirements.txt`, `Cargo.toml`)	One declared file listing the skills, prompts, instructions, agents, plugins, and MCP servers a project needs	PORTABILITY
Lockfile	Pins the exact resolved graph so restores match (`package-lock.json`, `Pipfile.lock`, `Cargo.lock`)	Exact resolved version and content identity of every primitive	REPRODUCIBILITY
Version pinning	Turns "some compatible version" into "this exact one"	Pin a primitive to a ref or commit instead of a moving branch	REPRODUCIBILITY
Sources / registries	Where a dependency resolves from (npm registry, PyPI, crates.io)	Where a primitive is fetched from	PROVENANCE / SECURITY
Provenance	Know where each artifact came from and that it is intact (integrity hashes, signatures)	Know the source *and* the content hash of each installed primitive	PROVENANCE / SECURITY
Repeatable restore	One command rebuilds the exact environment (`npm ci`, hashed `pip install`)	One command restores identical context on any machine and any harness	PORTABILITY + REPRODUCIBILITY

The six package-manager fundamentals, mapped onto agent context and the property each protects.

The lesson isn't the downloader

It is tempting to summarize a package manager as "a downloader with a registry." That misses the load-bearing part. Convenience was the hook; the durable contribution was the **manifest-plus-lockfile contract** — declared *intent* kept separate from a pinned, verifiable *result*. The registry is actually the most optional piece: npm, pip, and Cargo all resolve git and filesystem sources independently of any central index. A reader who remembers only "APM installs agent files" has taken the hook and dropped the lesson. The point is that context becomes *declared, pinned, and reviewable* like any other dependency.

Agent context is where application code was before lockfiles

Naming the *phase* is what makes the fix feel inevitable rather than imposed. Agent context today sits exactly where application dependencies sat before npm, pip, and Cargo: essential to the work, shared across a team, but neither declared nor pinned — managed by copying files, pasting from wikis, and per-machine setup. APM names this gap directly: agent context is "essential to work, shared by teams, but often not declared or pinned," and "there's no manifest for it" (microsoft/apm README). You have lived through the pre-lockfile era of application code and you know how it ended; that memory does the persuading.

Two cautions keep this honest. First, this is a **maturity-phase** claim, not an "APM is npm" claim — the category is genuinely young (SKILL.md, AGENTS.md, and MCP are still settling, and APM itself is pre-1.0 and ships often). Second, the common objection is "agent config is too new and too fluid to manage like a dependency." The historical counter is that application dependencies were *also* fluid and fast-moving before manifests existed: declaring context is what *creates* stability, not something you wait for stability to earn.

In APM

• MERIDIAN

Back at Meridian, you gather the six developers at a whiteboard rather than opening any files. On the left you write what everyone already trusts; on the right, its agent-context twin:

- `package.json` → `apm.yml` — the manifest we hand-author.
- `package-lock.json` → `apm.lock.yaml` — the lockfile a tool generates.
- `npm audit` → `apm audit` — a check we can run in CI.

"Nobody here would onboard by copying `node_modules` between laptops," you say. "We ship a `package-lock.json` and run one install. Agent context is the one dependency we still pass around by hand." The team nods — the fix stops feeling novel and starts feeling overdue, and for the first time they share a vocabulary for it. Then Anika, the new joiner, frowns at her laptop: "I searched 'apm install' last night and found *three* different tools. Which one is ours?" Good question — and the answer is the first pitfall of this chapter.

The same shape, named in APM

The whiteboard analogy lands because APM deliberately transplants the manifest-plus-lockfile shape onto agent-configuration files (What is APM?). Four names carry the four fundamentals you most need to recognize now — syntax and flags arrive in later chapters:

- `apm.yml` is the **manifest**: the human-authored declaration of intent. This is what carries • PORTABILITY.
- `apm.lock.yaml` is the **lockfile**: machine-generated, never hand-edited, pinning every dependency to an exact ref and content hash. This carries • REPRODUCIBILITY.
- `owner/repo#<ref>` is a **version pin**, where `<ref>` is a git tag, branch, or commit SHA. It is the git-native form of `pkg@1.2.3`, and it is what lets the lockfile promise byte-for-byte restores — • REPRODUCIBILITY again.
- **Git servers are the "registry."** APM resolves dependencies straight from git hosts, because "any git repository is a valid APM package" (What is APM?). Knowing the source is half of • PROVENANCE / SECURITY.

If you already think in npm, pip, or Cargo terms, the whole vocabulary transfers with one lookup:

You know...	...in APM
`package.json` / `requirements.txt` / `Cargo.toml`	`apm.yml`
`package-lock.json` / `Cargo.lock`	`apm.lock.yaml`
`pkg@1.2.3` version pin	`owner/repo#<ref>` (tag, branch, or SHA)
npmjs.com / PyPI / crates.io	git host (GitHub, GitLab, Azure DevOps, Bitbucket, Gitea, or a git URL)
`npm view` / `pip index versions`	`apm view <pkg> versions`
`npm outdated`	`apm outdated`
`npm update` / `pip install -U`	`apm update`
`npm audit`	`apm audit` (hidden-Unicode + lockfile integrity, *not* a CVE feed)
`npm install` (restore)	`apm install`

A cheat-sheet from what you already know to its APM counterpart (apm v0.23.1).

Notice the bottom rows: the maintenance verbs you reach for in npm all have APM counterparts. There is an `apm outdated` / `apm update` / `apm audit` trio, and `apm view <pkg> versions` lists the refs you can pin to. This chapter only *names* them so the analogy feels complete; Chapter 7 puts the lifecycle to work — including what `apm audit` does and does not check — and Chapter 8 covers the install-time security scan.

When to use / pitfalls

First, make sure it's the right "apm"

Anika's question is the one to answer before any tutorial: several unrelated tools share the name **apm** (and the near-homophone *aipm*). Run the wrong `pip install` or follow the wrong docs and you land on an entirely different product, with a different manifest, scope, and trust model. Getting identity right is a prerequisite to everything else in this book.

> ⓘ **The three-part tell for Microsoft's APM**
>
> The tool this book teaches has a fingerprint you can check in seconds. It is the `apm` you install with `pip install apm-cli` (PyPI), whose repository is `microsoft/apm`, and whose docs live at `microsoft.github.io/apm`. Its **three-file model** is `apm.yml`, `apm.lock.yaml`, and `apm-policy.yml`. If instead you see `apm-lock.json`, `aipm.toml`, or a hosted central registry as the primary identity, you are looking at a *different* tool.

Name	Owner	How you install / identify it	Fingerprint	What it actually is
Microsoft APM (*this book*)	Microsoft — `microsoft/apm`	`pip install apm-cli`; docs `microsoft.github.io/apm`	`apm.yml` + `apm.lock.yaml` + `apm-policy.yml`	The manifest-driven dependency manager for agent context taught here
`orthogonalhq/apm` (*same CLI name*)	Orthogonal (independent)	github.com/orthogonalhq/apm; registry `apm.orthg.nl`	`apm-lock.json` (no manifest)	An unrelated skills installer that *does* ship a public central registry
`TheLarkInn/aipm`	Sean Larkin (independent)	github.com/TheLarkInn/aipm	`aipm.toml`	A different "AI plugin manager" (Rust), for skills, agents, MCP, and hooks
Application Performance Monitoring	Many vendors	n/a	n/a	The same acronym for an unrelated observability discipline (metrics, traces)
Atom Package Manager	GitHub's retired Atom editor	the `apm` command	n/a	A historical namesake; packages for the Atom editor, nothing to do with agents

Telling the "apm" names apart — owner, how you identify it, and what it actually is.

The quickest proof they are separate projects, not forks of one, is the manifest split: `apm-lock.json` vs. `aipm.toml` vs. `apm.lock.yaml`. Community explainers publish dedicated disambiguation sections precisely because the confusion is real (a multi-way "APM" disambiguation).

Where the analogy breaks

The mapping is strong, but two asymmetries keep it honest — name both to a colleague so you don't over-claim.

> **⚠There is no central APM registry yet**
>
> APM borrows the manifest-plus-lockfile shape but not (yet) the other half of the classic story: a central, indexed registry with a search-and-publish API. A *source* is where a single dependency resolves from (a git repo at a ref); a *registry* is a central catalog with an HTTP API. APM has robust sources and pinning today, but no central registry — the registry HTTP API is explicitly deferred to v0.2 by the OpenAPM v0.1 spec. So when this book says "registry," the honest mapping today is **git servers**. This is a discovery/UX gap, not a correctness gap: reproducibility comes from the lockfile (a pinned commit plus a content hash), which delivers byte-for-byte restores *without* a central index.

> **⚠ `apm audit` is not `npm audit`**
>
> The cheat-sheet lines them up, but their jobs differ. As inspected on the v0.23.1 CLI, `apm audit` scans installed packages for **hidden-Unicode characters** and, with `--ci`, checks lockfile integrity. It is *not* a CVE vulnerability feed like `npm audit` or `pip-audit`. Promise a teammate an integrity and hidden-character check, not a security-advisory scan. Chapter 7 covers exactly what it does and doesn't catch (and Chapter 8 the install-time security scan).

A third honesty note is smaller but sets up Part III. When you run `apm install owner/repo` *without* a `#<ref>`, APM records the bare shorthand in `apm.yml` and tracks the source's default branch. That is convenient, but **not reproducible on its own** — tomorrow's install could resolve a newer commit. Pinning (`#<ref>`) plus the lockfile are what convert "works on my machine today" into "byte-for-byte, forever," a payoff Chapter 6 makes concrete.

> ◆ FOR ENGINEERING LEADERS
>
> Read the name collision as a governance signal, not trivia. "Install apm" in an onboarding script is ambiguous, and the *wrong* apm has a different trust and registry model — so the first governance act is simply standardizing on the canonical identity (`apm-cli`, `microsoft/apm`, `microsoft.github.io/apm`) in your setup docs. More broadly, what you are adopting here is **dependency governance, not IDE personalization**: you are deciding which sources your agents may pull context from and pinning them so they can be reviewed and audited — the same discipline you already require of `package.json` or `requirements.txt`, not a per-developer editor preference copied between laptops.

Worked example

The pin that makes it reproducible

Chapter 2 introduces no new Meridian manifest — that is deliberate; the team writes its first real `apm.yml` in Chapter 4. The one piece of syntax worth recognizing now is the version pin, so it looks familiar when it arrives: `owner/repo#<ref>`, where `<ref>` is a git tag, branch, or commit SHA. To pin the public sample package to a released tag:

Pin syntax — the git-native form of `pkg@1.2.3` (verified: tag `v1.0.0` resolves to a real commit on a remote git host).
apm v0.23.1 · ◆ NEEDS NETWORK

```
# owner/repo#<ref>  --  <ref> is a git tag, branch, or commit SHA
# Installs the public sample package pinned to exactly tag v1.0.0.
apm install microsoft/apm-sample-package#v1.0.0
```

And to discover which refs actually exist before you commit to one, `apm view` is the "versions" verb — the analogue of `npm view` / `pip index versions`:

List the remote tags and branches you can pin to (verified: contacts the remote and lists real refs). apm v0.23.1
NEEDS NETWORK

```
# Lists available remote tags and branches for the package.
apm view microsoft/apm-sample-package versions
```

Both commands reach a remote git host, so they are shown here as syntax rather than with captured output — Chapter 5 runs an install end to end, and Chapter 4 is where Meridian writes pins like these into its own manifest for the first time.

Recap & next

Recap

- Package managers normalized six moves — manifest, lockfile, pinning, sources, provenance, repeatable restore — and each has a one-for-one agent-context analogue that protects a Chapter 1 property.
- The load-bearing lesson is the **manifest-plus-lockfile contract** (declared intent vs. pinned result), not the downloader or the registry — the registry is the most optional part.
- Agent context is in the pre-package-manager phase application code was in before lockfiles; that is a maturity-phase claim, and declaring context is what *creates* stability.
- In APM the shape has names: `apm.yml` (manifest), `apm.lock.yaml` (lockfile), `owner/repo#<ref>` (pin), and git servers as the "registry." The `outdated`/`update`/`audit` trio and `apm view … versions` are named here and taught later.
- Verify identity with the three-part tell (`apm-cli` + `microsoft/apm` + `microsoft.github.io/apm`, three-file model) so you don't install a different "apm."
- Two honesty notes: "registry" maps to git servers today (central registry deferred to OpenAPM v0.2), and `apm audit` is an integrity / hidden-Unicode scan, not a CVE feed.

Next

You now share the team's vocabulary for *what* APM manages. Next, learn what those managed things actually are: Chapter 3 — Primitives & Harnesses names the primitives (skills, prompts, instructions, agents, plugins, MCP servers) and shows how each maps onto the harnesses that read them.

PART I · WHY CONTEXT NEEDS A MANIFEST / CHAPTER 3

Primitives & Harnesses

Know the vocabulary of what APM manages and how those primitives relate to agent harnesses.

Objective

After this chapter you can name the seven primitive types APM manages — instruction, prompt, skill, agent, plugin, MCP server, and hook — tell the three that look alike apart by *who triggers them*, explain that APM is built *on* the open AGENTS.md, SKILL.md, and MCP standards rather than replacing them, and describe how one source primitive compiles into each harness's native files. Naming the primitives is what makes a context declaration PORTABLE and a security decision GOVERNABLE in the chapters ahead.

Concept/Theory

You can't package what you can't name

In Chapter 1 the team's agent context was an undifferentiated pile — "an AI setup," "some rules," "that prompt." You can't declare a pile in a manifest, pin it in a lockfile, or gate it in a policy. Before any of that, each item needs a *type*. That is the whole job of this chapter: hand you the nouns. Chapter 2 showed that names and namespaces are what tame ambiguity in package managers; here we give agent context its names.

APM's word for a typed unit is a **primitive** — "the atomic unit APM ships." The supported kinds are instructions, skills, prompts, agents, hooks, commands, plugins, and MCP servers (glossary) — though "command" is a *prompt* as some harnesses project it rather than a separate thing to author, and the experimental `canvas` type is out of scope here, which leaves **seven** authorable types. A primitive is not the same as a *package*: a package is the unit of distribution — a directory with an `apm.yml` that can bundle many primitives — while a primitive is one typed unit inside it. And not every file is a primitive: only files under the recognised primitive layout are discovered, validated, and deployed; everything else is just repo content (glossary).

The primitive vocabulary

APM manages seven primitive types you'll meet daily. The point of the vocabulary is that they are *not* interchangeable: each has a different trigger, cost, and blast radius. Read the middle column carefully — *who* (or *what*) *invokes* it is the axis that keeps the types straight.

Primitive	What it is	Who invokes it	Problem it solves
Instruction	Repository-scoped guardrails and coding standards, scoped by file glob	*Nobody* — ambient, read on every turn (always-on)	Persistent rules you never have to re-state
Prompt	A parameterized, runnable workflow — "a callable program for an LLM"	*You* — explicitly, by name (you run it)	A repeatable multi-step task or checklist on demand
Skill	Model-invocable know-how — a `SKILL.md` folder plus bundled resources	*The model* — when a task matches its description (it reaches for it)	Specialized capability loaded only when relevant
Agent	A persona with its own scope, tools, and system prompt	The user or orchestrator selects it as the acting persona	A bounded specialist instead of one generalist
Plugin	A bundle of the primitives above, packaged for one-shot install	Installed as one unit, then unpacked into its primitives	Shipping a set of primitives together
MCP server	An external tool or data connection over Model Context Protocol	The agent at runtime, when it calls that tool	Governed access to outside tools and data
Hook	A lifecycle handler that runs before or after agent tool calls	The harness runtime, automatically at that event	Deterministic automation around tool calls

The seven primitive types APM manages, by trigger. Definitions condensed from APM's "what APM manages" table and primitive catalogue (What is APM?, primitives and targets).

> **ⓘ The one distinction to get right: instruction ≠ prompt ≠ skill**
>
> All three are "markdown that steers the agent," so they blur together. They differ by *who triggers them* and *what shape they take*. An **instruction** is never invoked — it is ambient guidance read on every turn inside its `applyTo` glob. A **prompt** is invoked by a human, by name — a parameterized workflow, "a callable program for an LLM." A **skill** is invoked by the model, autonomously, through *progressive disclosure*: the agent sees only the skill's name and description until a task matches, then loads the full `SKILL.md` and any bundled scripts. The mnemonic to keep: *instruction = always on; prompt = you run it; skill = the model reaches for it* (primitives and targets).

Two more pairs are worth keeping straight. An **agent** is *who* is acting — a persona with its own tools and boundaries; a **skill** is *how* to do a task — a procedure the acting agent can pull in, and one agent can use many skills. And a **plugin** is not a new capability at all: it is a packaging format that ships several primitives together and is normalized back into those same typed primitives at install time (primitives and targets).

Built on standards, not instead of them

APM does not invent its file formats. It is "built on standards: AGENTS.md · Agent Skills / SKILL.md · MCP" (microsoft/apm README). Each standard is an independent, open format APM adopts for one slice of the vocabulary:

- **AGENTS.md** — "a simple, open format for guiding coding agents… a dedicated, predictable place to provide context and instructions" that works across many agents (agents.md). APM's *instructions* compile into AGENTS.md-style files.
- **Agent Skills / SKILL.md** — "a lightweight, open format for extending AI agent capabilities… a skill is a folder containing a `SKILL.md` file" loaded by progressive disclosure (agentskills.io). APM's *skill* primitive **is** this format.
- **MCP (Model Context Protocol)** — "an open-source standard for connecting AI applications to external systems" (modelcontextprotocol.io). APM declares MCP *servers* against this protocol.

Standing on shared standards is what makes APM a *manager* rather than another silo. The standards define the file and protocol *formats*; APM adds the dependency *layer* on top — resolution, pinning, a lockfile, policy, and multi-harness compilation. So APM-produced context is readable by tools that never installed APM, and APM can consume primitives authored by people who never used it. Concretely, `apm compile` emits distributed AGENTS.md context files from your instructions — the open standard is the output, not a proprietary blob (What is APM?).

• MERIDIAN

Armed with the vocabulary, you sort Meridian's scattered context from Chapter 1 into a **primitive map** — every item finally gets a type instead of a shrug:

- the checkout backend guardrails ("use `Money`, never log a card number") are an **instruction** — always on;
- the fraud-review PR checklist is a **prompt** — a reviewer runs it by name;
- the secure-review know-how is a **skill** — the model reaches for it on payment-touching diffs;
- the API-architect persona is an **agent** — a specialist that owns API-shape decisions;
- the payment-sandbox tool is an **MCP server** — declared and gated, not shipped.

Two of the seven types don't appear yet: Meridian has no **plugin** (it isn't shipping a bundle to other teams) and no **hook** (no lifecycle automation) — and that's fine. This map is the seed of Chapter 4's manifest, but it is *not yet a manifest*: it's just the pile, finally typed.

In APM

One source, many harnesses

A **harness** is the agent runtime that executes primitives — harnesses include GitHub Copilot, Claude Code, Cursor, Codex, Gemini, OpenCode, Windsurf, and Kiro, with Antigravity as an additional explicit-only target — and "each harness has its own primitive directory layout and file format" (glossary). A primitive, by contrast, is harness-agnostic. APM's central move is to sit between the two: you author one set of primitives under `.apm/`, and APM writes them into each harness's native location. It "does not invent a runtime format. It writes the files each tool already understands and stays out of the way at agent runtime" (What is APM?).

Because each harness has its own format, APM either writes a primitive *natively* (the harness reads it directly) or *compiles* it (APM transforms it into a shape the harness understands). The clearest proof is to install *one* source package into two harnesses and look at what lands on disk. The public `microsoft/apm-sample-package` ships four of the seven primitive types, so four rows follow — a real install, same source, zero edits, once with `--target copilot` and once with `--target claude`:

Source primitive (`.apm/…`)	Copilot (`.github/…`) — native	Claude Code (`.claude/…`) — compiled
instruction `instructions/*.instructions.md`	`.github/instructions/*.instructions.md`	`.claude/rules/*.md` — renamed to a **rule**
prompt `prompts/*.prompt.md`	`.github/prompts/*.prompt.md`	`.claude/commands/*.md` — a **/command** (`mode` key dropped)
agent `agents/*.agent.md`	`.github/agents/*.agent.md`	`.claude/agents/*.md`
skill `skills/<name>/SKILL.md`	`.agents/skills/<name>/SKILL.md`	`.claude/skills/<name>/SKILL.md`

One source primitive set → two harnesses, two on-disk layouts — and even two different *names*. Observed from a real install of `microsoft/apm-sample-package`. `apm v0.23.1`

Read across any row and portability stops being a slogan. The same source instruction becomes a Copilot `.github/instructions/` file but is renamed to a Claude `rule`; the same prompt is a native Copilot prompt but a compiled Claude `/command`. (Skills for the Copilot family land in the cross-client `.agents/skills/` home rather than a Copilot-only directory, which is why that row's left cell reads `.agents/`, not `.github/`.) The intent is identical everywhere; the file is whatever each harness natively reads.

> ⚠**Pitfall: "compiled" does not mean byte-identical**
>
> One source primitive lands as whatever each harness natively reads — not the same file everywhere. The install above didn't just relocate files; it *renamed* an instruction to a Claude *rule* and a prompt to a */command*, and it **dropped** the `mode` frontmatter key because Claude commands don't support it, warning as it did. That is *compiled, by design*: the intent is preserved, the bytes are not. When a harness can't represent a primitive at all, the compatibility matrix marks that cell `unsupported` rather than inventing a lossy shim (primitives and targets). Expect parity to be *high but bounded*, and treat the matrix as the source of truth.

Targets are the declaration; the harness is the runtime

You don't name a harness directly in the manifest — you list **targets**. A target is "the declaration... the manifest or CLI selector for which harnesses to compile for," while "the harness is the runtime that consumes the compiled output" (glossary). Eight harnesses are configured as targets by default — `copilot`, `claude`, `cursor`, `codex`, `gemini`, `opencode`, `windsurf`, and `kiro` — which is why this book says "many harnesses" and lists them rather than promising every tool ever built (What is APM?). Selecting and pinning targets is Chapter 5's job; here it is enough to see the shape: one declaration, many native destinations.

Two of the book's four properties are already visible in this layer. Declaring context as typed primitives is the basis of PORTABILITY — one source set, compiled into every harness you target. And because each primitive is a named, typed unit, an organization can allow or deny it *by category*, which is what makes GOVERNANCE discussable at all. The remaining two arrive downstream: the lockfile pins each deployed primitive by content hash — the basis of REPRODUCIBILITY (Chapter 6) and of PROVENANCE (Chapter 8).

> ◆ FOR ENGINEERING LEADERS
>
> The vocabulary is what makes governance concrete. "Approve the team's AI setup" is an opaque request; "*approve this* ***instruction****, this* ***prompt****, this* ***skill****, and this* ***MCP server****, from these sources*" is a reviewable one. Typed primitives turn an undifferentiated blob into line items a security team can allow or deny individually — and because an MCP server is now a named, declared unit, you can finally ask "which external tools can our agents reach, and who approved each?" and get an answer. GOVERNANCE becomes a checklist rather than a leap of faith; the policy that enforces it arrives in Chapter 9.

When to use / pitfalls

Which primitive for the job?

Distinct names prevent category errors. "Add a security review" is ambiguous until you say *who triggers it*: a persistent *instruction* that always warns about logging card data, a *prompt* the reviewer runs on a PR, a *skill* the model pulls in when it recognises a payment change, or an *agent* that owns review end to end. Each has a different cost and blast radius, so pick by trigger:

If you want...	Reach for a...	Because
a rule enforced every turn that no one has to remember	**instruction**	it's ambient and always on
a named checklist a human kicks off on a PR	**prompt**	you run it, explicitly, by name
know-how the model should pull in only when a task matches	**skill**	the model reaches for it, keeping context lean
a bounded specialist that owns a task end to end	**agent**	it's a persona with its own tools and scope
deterministic automation around every tool call	**hook**	the runtime fires it at a lifecycle event
a governed connection to an external tool or dataset	**MCP server**	it's declared and gated, run elsewhere
several of the above shipped as one installable unit	**plugin**	it's a packaging format, unpacked into primitives

Choosing a primitive by who — or what — triggers it.

> **⚠ Pitfall: declaring an MCP server is not running it**
>
> An MCP server is the one primitive APM *manages but does not ship*. "MCP servers are external processes; APM declares and gates them but does not ship their code" (glossary). APM writes each harness's MCP configuration and applies policy to the connection; the server itself runs elsewhere, as its own process. So "we declared the payment-sandbox MCP server" means the connection is wired and gated — not that APM launched a sandbox. The gating that makes this safe, including withholding undeclared transitive MCP servers, is the subject of Chapter 8.

> **ⓘ Two words that aren't primitives: *command* and *canvas***
>
> You'll see "command" and "canvas" near APM, but neither is one of the seven types you author. A **command** is just a *prompt* projected into a harness that names it differently — the same `.prompt.md` source becomes Copilot's prompt and Claude's `/command`, with no separate commands directory to maintain. **Canvas** is an experimental type, out of scope for this book. A **plugin**, by contrast, *is* one of the seven — the *packaging* primitive, authored in its own right with `apm plugin init`, that bundles several primitives together and is normalized back into those same typed primitives at install time. So the count holds: seven types to author, with *command* a projection of a prompt and *canvas* left out as experimental (primitives and targets).

Worked example

Meridian's context, finally typed

Here is Meridian's pile from Chapter 1, sorted into its primitive map. Nothing below is a manifest yet — it is the same inventory, now with a type and a trigger on every row. Read down the trigger column and the instruction/prompt/skill split snaps into focus: three of these are markdown, but one is always on, one you run, and one the model reaches for.

Meridian's primitive map — every scattered file from Chapter 1, now typed. A map, not yet a manifest. apm v0.23.1

```
# meridian-checkout, now TYPED:  source item  ->  primitive   (who triggers it)
#
checkout backend guardrails (use Money) -> instruction  nobody: read every turn (always-on)
fraud-review PR checklist               -> prompt       you: invoked by name, on demand
secure-review know-how                  -> skill        the model: reached for on payment diffs
API-architect persona                   -> agent        selected as the acting specialist
payment-sandbox tool connection         -> mcp server   the agent at runtime (declared + gated)
#
# not yet needed: plugin (no bundle to ship) . hook (no lifecycle automation)
```

That map is heading toward a manifest, but guard the distinction: it is *not yet a manifest*. The shape it will take groups the five rows into two dependency buckets — `apm` for primitives and `mcp` for MCP servers — plus the harness `targets` to build for. Everything below is illustrative; Chapter 4 validates the real syntax and pins each source, and Chapter 5 installs it.

Conceptual only — the same map as a manifest *shape*, showing the two dependency buckets and harness targets. Not validated until Chapter 4. apm v0.23.1

```
name: meridian-checkout-agent-context
targets: [copilot, claude, cursor]   # 3 of APM's 8 default harness targets
dependencies:
  apm: []              # primitives -- declared for real in Chapter 4:
    #  - <checkout-domain instruction>
    #  - <fraud-review prompt>
    #  - <secure-review skill>
    #  - <api-architect agent>
  mcp: []              # MCP servers -- declared + gated here, the server runs elsewhere:
    #  - <payment-sandbox>
```

Recap & next

Recap

- A **primitive** is APM's atomic, typed unit of agent context. Naming the primitives is the prerequisite for everything else — you can't package, pin, or govern what you can't name.
- There are seven types. The three that look alike differ by trigger: *instruction = always on; prompt = you run it; skill = the model reaches for it*. An **agent** is who acts; a **plugin** is packaging; an **MCP server** is an external tool; a **hook** is lifecycle automation.
- APM is *built on* open standards — AGENTS.md, SKILL.md, and MCP — not a replacement for them; `apm compile` emits distributed AGENTS.md files.
- One source primitive compiles into each harness's native files: an instruction becomes a Copilot instruction or a Claude *rule*; a prompt becomes a Copilot prompt or a Claude */command*. Compiled is not byte-identical; parity across the eight default harnesses is high but bounded (native / compiled / unsupported).
- **Targets** are the declaration you list; the **harness** is the runtime that consumes the output.
- Typed primitives underpin PORTABILITY and make GOVERNANCE a concrete checklist. Meridian typed its pile into a primitive map — the seed of the next chapter's manifest, but not yet a manifest.

Next

You now have the nouns and a typed map. Chapter 4 — The Manifest: apm.yml turns Meridian's primitive map into a valid `apm.yml`: real dependency syntax, pinned git sources, and the two buckets (`apm` and `mcp`) filled in for the

first time.

PART II · PORTABLE BY MANIFEST / CHAPTER 4

The Manifest: apm.yml

Author a valid apm.yml that declares Meridian's first shared agent-context dependencies.

Objective

After this chapter you can author a valid `apm.yml` — the first real APM file you write — that declares a project's identity, the harnesses it targets, and one locally-authored instruction, and explain why that single committed file is what turns Chapter 1's scattered, per-machine setup into ● PORTABLE, reviewable, version-controlled state.

Concept/Theory

A manifest is "what this project needs," written down

Before a manifest, a project's agent context is whatever files happen to sit on each machine — the exact drift Chapter 1 diagnosed at Meridian: a `copilot-instructions.md` here, a private `review.prompt.md` there, Cursor rules no one else can see. Nothing declares what the *project* needs; each laptop is its own source of truth, so the setups quietly diverge. A **manifest** replaces "whatever is on my disk" with "what this repository declares" — one version-controlled file that travels with the code. This is where portability stops being a slogan and becomes a file you can open in a pull request.

In APM that file is `apm.yml`, and the docs describe it as a *contract*: it "declares the full closure of agent primitive dependencies, MCP servers, scripts, compilation settings … for a project" and is "the contract between package authors, runtimes, and integrators" (manifest schema). Only two fields are required to parse — `name` and `version` — and everything else is optional (manifest schema §2). A valid manifest can be three lines long (package anatomy).

Chapter 1 ended with a *sketch* of Meridian's `apm.yml` and called it "a promise, not yet a program." This chapter makes it real: the same shape, now a file the tool actually reads. If you have ever opened a `package.json`, you already know how to read it — "the shape mirrors `package.json` on purpose: `name`, `version`, `dependencies`… `scripts`" (package anatomy).

What the manifest is, and what it is not

Two distinctions keep the rest of the book straight. First, a manifest is *not* the per-machine files it replaces. Those were many hand-edited *outputs*; the manifest is one authored *input*, committed to the repo, that those outputs are generated from. You edit the declaration — never the deployed copies under `.github/`, `.claude/`, or `.cursor/`, which APM regenerates on install.

Second, a manifest is *not* a lockfile. The manifest is **human-authored declared intent** — *what* you want, which may include a moving branch or a version range. The lockfile (`apm.lock.yaml`, Chapter 6) is the **machine-generated resolved state** — the exact commits and content hashes you actually got. The manifest says *what*; the lockfile records *exactly which* (package anatomy). This chapter is entirely about the first file.

The manifest is the ● PORTABILITY surface — its whole job — but it also *seeds* the other three properties without yet delivering them: it is the input the lockfile resolves (● REPRODUCIBILITY, Chapter 6), the surface a policy inspects (● GOVERNANCE, Chapter 9), and the declaration that makes provenance reviewable (● PROVENANCE / SECURITY, Chapter 8). Get the declaration right here and the other three chapters have something honest to work with.

> **ⓘ Misconception: "`apm.yml` is just config for the CLI"**
>
> It is not tool-local configuration. `apm.yml` is the project's shared, harness-agnostic contract, and any conforming resolver — not only the `apm` CLI — can consume it "to install, compile, run, and pack agentic workflows" (manifest schema, abstract). It belongs to the repository, committed next to your code, not to a workstation.

> • MERIDIAN
>
> The turn this chapter is small and deliberate. The team stops maintaining three private setups and commits *one* `apm.yml` at the root of `meridian-checkout`. Nothing about their context is new — it is the same primitive map from Chapter 3 — but for the first time it is *declared* instead of *scattered*. The team's very first manifest depends on no one else's packages: it just makes their own checkout rule portable across Copilot, Claude Code, and Cursor. That single committed file is the whole point of the chapter.

In APM

What `apm init` writes, key by key

You do not start from a blank file. `apm init` "creates exactly one file — the manifest" (your first package), then you edit it. Run in an empty directory, it emits sensible defaults — and three of them surprise readers who hand-wrote a sketch first, so read the caption closely:

The shape `apm init` generates in an empty `acme-agents` directory — `name` from the directory, `author` from your git config. Note the identity line is `author:` (there is *no* `license:` key), `targets:` is *commented out*, and `includes` defaults to `auto`. `apm v0.23.1`

```
# apm.yml -- one file, sensible defaults; you edit it from here
name: acme-agents
version: 1.0.0
description: APM project for acme-agents
author: Ravi Sharma           # auto-detected from your git user.name
# Which agent platforms to deploy to (uncomment to pin):
# targets:
#   - copilot
#   - claude

dependencies:
  apm: []
  mcp: []
includes: auto
scripts: {}
```

Every key here is a question the team used to answer per-machine, now answered in one reviewable place. The point is the *meaning and shape* of each field, not an exhaustive flag tour:

Key	Type	What it declares	Carries
`name`	string	The package's identity — its slug when another repo depends on it. Defaults to the directory name.	PORTABILITY
`version`	string (semver)	This package's own version. Meridian starts pre-1.0 at `0.1.0` and advances one beat per chapter.	PORTABILITY
`description`	string	A one-line human summary shown in listings. Replace the boilerplate.	PORTABILITY
`author`	string	Optional attribution. `apm init` auto-fills it from your git `user.name`; set it to the team or org for a shared repo.	PORTABILITY
`targets`	list (*commented* by default)	Which harnesses to deploy to. **Absent means APM auto-detects** from the repo; uncomment and pin for a shared repo.	PORTABILITY
`dependencies.apm`	list	External primitive packages, git-sourced and pinned by ref. Empty is valid. Filled for real in Chapter 5.	SEEDS REPRODUCIBILITY
`dependencies.mcp`	list of *objects*	MCP server declarations (external tools). A list *of objects*, not a name→map. Declared and blocked in Chapter 8 (security), policy-governed in Chapter 9.	SEEDS SECURITY / GOVERNANCE
`includes`	`auto` | list	Which local `.apm/` primitives to compile and deploy. `auto` discovers everything under `.apm/`.	PORTABILITY
`scripts`	map	Named shell commands (`name: "command"`) run with `apm run`. Empty for now; used in Chapter 5.	PORTABILITY

The keys `apm init` writes, at a glance — what each declares, its type, and the property it carries. Field set confirmed against a fresh `apm init`. `apm v0.23.1`

Two of those keys read straight out of Chapter 3. `targets` is the *target ≠ harness* distinction, now written down: you *declare* the targets, and the harness still runs independently. Pinning them in the manifest makes the deployed output deterministic "for every developer, CI runner, and cloud agent," rather than silently tracking whichever tool folders happen to exist on the last person's machine (manifest schema §3.6). And `includes: auto` is what connects the manifest to the primitives you author locally, discussed next.

`.apm/` is where you author; `apm_modules/` is what you install

A manifest describes two sources of primitives that look different but land in the same place. **Locally-authored** primitives are files your repo writes itself, under the `.apm/` source tree (`instructions/`, `prompts/`, `skills/`, `agents/`, `hooks/`) — "the conventional source root for APM packages." **External dependencies** are primitives pulled from other repositories, declared under `dependencies` and fetched at install time into the generated, gitignored `apm_modules/`. You edit `.apm/`; you never hand-edit `apm_modules/` (package anatomy).

`includes: auto` is the bridge for the first kind: with no remote dependencies declared at all, `apm install` simply "walks your local `.apm/` tree and deploys what it finds" — "APM treats your repo as the package and deploys its `.apm/` content" (your first package). That is precisely Meridian's first manifest: one local instruction, zero remote dependencies. Consuming other people's packages (Chapter 5) is an *addition* to local authoring, never a precondition for it.

The two dependency buckets — shape now, installs in Chapter 5

Meridian declares no dependencies at `v0.1.0`, but you should recognize the two buckets when you meet them. `dependencies.apm` is a list of git-sourced package refs; `dependencies.mcp` is a list of MCP server *objects*. Here is the shape only — do not add either to your first manifest:

Shape only: the two dependency buckets. Meridian declares neither at `v0.1.0` — remote packages arrive in Chapter 5, MCP servers in Chapter 8. `apm v0.23.1`

```
dependencies:
  apm:
    - microsoft/apm-sample-package#v1.0.0     # owner/repo[/subpath]#<ref> -- pin the ref
  mcp:
    - name: fetch                             # each entry is an OBJECT, not a name -> map
      registry: false
      transport: stdio
      command: npx
      args: ["-y", "@modelcontextprotocol/server-fetch"]
```

Pinning the git ref (`#v1.0.0`) is what keeps a restore deterministic and a diff honest — the reproducibility lesson from Chapter 2, enforced here by convention. The subpath form (`owner/repo/skills/<name>`) selects a single primitive from a monorepo. Both are declared here and *installed* for real in Chapter 5.

◆ FOR ENGINEERING LEADERS

This is the moment agent context enters code review. Because the manifest is a small, version-controlled text file, every change to it is a **diff** — and a diff is something a teammate can review in a pull request. "Added `microsoft/apm-sample-package#v1.0.0`," "targeted `cursor`," "moved a source from a branch to a tag" are now visible, reviewable edits rather than invisible per-machine changes. A pinned ref shows whether a dependency is immutable or moving; the source shows *whose* code it is. The accountability gap from Chapter 1 — security could not say which tools were installed or whether a prompt came from an approved source — closes the day these answers live in a committed file. The question to put to your team: should adding or changing `apm.yml` require review the same way changing `package.json` does? The policy layer that *enforces* the answer is Chapter 9; the reviewable diff is what makes that enforcement possible.

When to use / pitfalls

Keep your first manifest minimal

The dividing line from Chapter 1 still holds: commit an `apm.yml` the moment context is *shared* and *plural* — multiple developers, multiple harnesses, onboarding that should be reproducible. For a spike you will throw away, a manifest is ceremony you do not need yet.

When you do write one, resist the urge to fill every key. A strong first manifest declares only three things: **identity** (`name`, `version`, a real `description`), **targets** (the harnesses your team actually uses), and the **local primitives** it authors under `.apm/` (via `includes: auto`). Remote dependencies and scripts are real features — they are just Chapter 5's beat, added when you need them, not scaffolding to fill in up front. The empty `dependencies` buckets and `scripts: {}` that `apm init` leaves behind are inert and harmless; keep them and move on.

⚠ **Pitfall: there is no `license:` key — the identity line is `author:`**

A hand-written sketch often reaches for `license: MIT`, but `apm init` does not generate a `license` field. The attribution line it writes is `author:`, auto-filled from your git `user.name`. To stay faithful to a real manifest, use `author:` and omit `license:`. (A `license` value is harmless if you add it deliberately — it is simply not part of the generated shape.)

> ⚠ **Pitfall: `includes: auto` is not a restrictive allowlist**
>
> It is tempting to read an explicit `includes:` path list as "only ship these files." It is not an allowlist. Tested on `apm v0.23.1`: with a second, *unlisted* primitive present under `.apm/`, `apm install` deployed **both** primitives anyway. Use `includes: auto` — it matches what `apm init` writes, needs no upkeep, and auto-discovers everything under `.apm/`, so it will not mislead you into thinking you have scoped a deployment you have not.

> ⓘ **Two shape gotchas to internalize**
>
> **Commented `targets:` means auto-detect.** Left commented, APM detects targets from the repo's tool folders — convenient locally, but non-deterministic across machines and CI. For a shared repo, *uncomment and pin* the set (Meridian: `copilot`, `claude`, `cursor`).
>
> **`dependencies.mcp` is a list of objects, not a name → map.** Each entry is a `{ name, registry, transport, … }` object (see the shape above), which is what the CLI actually writes — not a `fetch: { command, args }` mapping. You will declare one for real in Chapter 8.

Worked example

Meridian's first real `apm.yml`

Here is the payoff. Meridian takes the `apm init` scaffold, sets its identity, uncomments and pins three targets, and authors one local instruction — the checkout-domain rule that drifted in Chapter 1 ("use the `Money` value object; treat retries as idempotent"). No remote dependencies, no scripts. This is the whole file:

Meridian's first real manifest (`v0.1.0`): identity, three targets, and — via `includes: auto` — one local instruction. Verified end to end with `apm install`; no network. `apm v0.23.1`

```yaml
name: meridian-checkout-agent-context
version: 0.1.0
description: Shared AI-agent context for the Meridian checkout service.
author: Meridian Checkout Team
targets:
  - copilot
  - claude
  - cursor

dependencies:
  apm: []
  mcp: []
includes: auto
scripts: {}
```

The instruction itself is authored once, under a recognized primitive directory, so `includes: auto` discovers it with zero extra configuration. Its frontmatter carries a `description` and an `applyTo` glob — the same instruction shape from Chapter 3:

`.apm/instructions/meridian-checkout.instructions.md` — one source of truth for the checkout rule, authored by hand. apm v0.23.1

```
---
description: Meridian checkout service engineering rules
applyTo: "**/*.{ts,tsx,md}"
---

Use the Meridian `Money` value object for currency math. Treat checkout retries as
idempotent operations keyed by `paymentAttemptId`. Never suggest storing card data in
application logs.
```

With the manifest and the one source file in place, a single command restores the context. Running `apm install` with *no package argument* means "deploy from `apm.yml`": APM treats the local project as one package and fans its instruction out to the declared targets.

`apm install` deploying the local instruction to all three targets — no network, no package fetched. The local project is the package. apm v0.23.1

```
$ apm install
  [i] Targets: claude, copilot, cursor  (source: apm.yml)
    [+] <project root> (local)
    |-- 3 rule(s) integrated -> 3 targets
  [i] Added apm_modules/ to .gitignore
  [*] Installed 1 APM dependency in 2.5s.
```

This is the concrete portability win. One source file, one command, and the same intent lands in three harnesses' native locations — different directories, different frontmatter keys, even a different file extension, with *zero* per-harness edits:

Harness	Deployed path	Frontmatter	Transform vs. your source
copilot	`.github/instructions/meridian-checkout.instructions.md`	`description` + `applyTo`	**unchanged** — native `.instructions.md`
claude	`.claude/rules/meridian-checkout.md`	`paths` only	`applyTo` → `paths`; `description` dropped; `.md`
cursor	`.cursor/rules/meridian-checkout.mdc`	`description` + `globs`	`applyTo` → `globs`; `description` kept; `.mdc`

One authored source → three native harness files. Observed on disk after `apm install`; frontmatter shown as APM actually projected it. apm v0.23.1

The transforms are exactly Chapter 3's "compiled is not byte-identical" rule, now proven on Meridian's *own* content. Read the two projected files side by side and the point is undeniable — same rule, each in its harness's native shape:

`.claude/rules/meridian-checkout.md` — `applyTo` became `paths`, and `description` was dropped. apm v0.23.1

```
---
paths:
  - "**/*.{ts,tsx,md}"
---

Use the Meridian `Money` value object for currency math. Treat checkout retries as
idempotent operations keyed by `paymentAttemptId`. Never suggest storing card data in
application logs.
```

`.cursor/rules/meridian-checkout.mdc` — `applyTo` became `globs`, `description` was kept, and the extension is `.mdc`. `apm v0.23.1`

```
---
description: Meridian checkout service engineering rules
globs: "**/*.{ts,tsx,md}"
---

Use the Meridian `Money` value object for currency math. Treat checkout retries as
idempotent operations keyed by `paymentAttemptId`. Never suggest storing card data in
application logs.
```

Two side effects are worth naming so they are not mistaken for drift. `apm install` auto-appended `apm_modules/` to `.gitignore` (it is generated, never committed), and it wrote an `apm.lock.yaml` recording the deployed files and their content hashes. With only local primitives, that lockfile is short and has no `dependencies` — the full lockfile treatment is Chapter 6. What matters here is what you authored: one manifest, one instruction, and a reviewable file the whole team now shares.

Recap & next

Recap

- A **manifest** turns "what this project needs" into reviewable, version-controlled state. In APM that file is `apm.yml` — the ● PORTABILITY surface, and the input that seeds reproducibility, governance, and provenance in later chapters.
- You author one *input*; APM generates the *outputs*. The manifest is declared intent (*what*); the lockfile is resolved state (*exactly which*, Chapter 6).
- `apm init` writes one file with sensible defaults: identity (`name`/`version`/`description`/`author`), commented `targets`, two empty `dependencies` buckets, `includes: auto`, and `scripts: {}`. Only `name` and `version` are required to parse.
- `includes: auto` auto-discovers your locally-authored primitives under `.apm/`; it is *not* a restrictive allowlist. You edit `.apm/`; APM generates `apm_modules/` and the per-harness output.
- Mind the corrections: the identity line is `author:`, not `license:`; commented `targets:` means auto-detect (pin it for a shared repo); and `dependencies.mcp` is a list of *objects*.
- Meridian committed its first real manifest (`v0.1.0`): identity, three targets, and one local instruction that `apm install` deployed to Copilot, Claude Code, and Cursor — one source, three native files, zero per-harness edits.

Next

Meridian's manifest declares its own context beautifully — but it depends on no one else yet. Chapter 5 — Install & Restore fills the `dependencies.apm` bucket with a real, pinned git source, runs `apm install` to consume an external package for the first time, and turns the empty `scripts` map into runnable commands — the everyday *install and restore* loop.

PART II · PORTABLE BY MANIFEST / CHAPTER 5

Install & Restore

Install and restore a project's agent context with the daily APM consumer loop.

Objective

After this chapter you can run APM's four-command consumer loop — `apm init`, `apm install <pkg>`, bare `apm install`, and `apm run <script>` — and tell its two install jobs apart: *add* (grow the manifest and re-lock) versus *restore* (reproduce exactly what is already declared). You will read the lockfile entry an install writes, know why a no-target install stops with exit `2` instead of guessing a harness, and know why `apm run` needs a runtime CLI on your `PATH`. This is the operation that turns Chapter 4's declaration into identical on-disk context in every developer's tools — the moment PORTABILITY stops being a promise and becomes a fact.

Concept/Theory

A manifest is a recipe, not a meal

Chapter 4 gave Meridian an `apm.yml`. On its own, that file changes nothing about what an agent sees: no instruction is in force, no prompt is available, no MCP server is wired up. A manifest is a *declaration* of what should exist; something still has to make it exist. **Install is that something.** It "resolves the dependencies declared in `apm.yml`, downloads them … and deploys the resulting primitives plus the project's own `.apm/` content into every harness target it detects. It writes `apm.lock.yaml` so the next install on any machine reproduces the same files" (apm install).

We call this move **materialization**: turning the declared state (`apm.yml` plus its lockfile) into real files each tool reads natively. It is where the book's first property becomes observable. Chapters 1–4 argued that scattered, hand-copied context *drifts*; a manifest only ends that drift if there is a single, mechanical way to realize it identically everywhere — "the next contributor runs bare `apm install` and gets the same bytes, pinned by commit and content hash" (Install packages). Editing the manifest without installing leaves the declared and materialized states out of sync — the very gap Chapter 6 formalizes.

Install is not "just downloading packages," either. It is a fail-closed pipeline — *resolve*, *policy gate*, *scan*, *integrate*, *lockfile* — and a security scan (plus an org policy gate, if present) runs "before writing anything to disk" (Install packages). That gate is Chapter 8's subject; here it is enough to see install as *gated deployment*, not a copy. And notice the by-product: every install writes `apm.lock.yaml`, which seeds REPRODUCIBILITY — the lockfile itself is Chapter 6, so for now just notice it appear.

The daily loop: four commands, one habit

APM's everyday surface is deliberately tiny. The consumer ramp names "the four commands you'll use almost every day," then draws the line: "That's the loop. Everything else is either lifecycle automation (`update`, `outdated`, `audit`) or a workflow extension" (Use APM packages). The point is a *habit*, not a flag reference — four verbs that map onto four real moments in a team's week:

Command	Job	The everyday moment (Meridian)
`apm init`	scaffold the manifest — **one-time** per project	when the manifest was born in Chapter 4
`apm install <pkg>`	**add** a dependency and re-lock	Lena needs a new shared skill
`apm install`	**restore** from the lockfile	Anika clones a fresh repo
`apm run <script>`	run a declared script (*experimental*)	anyone runs the checkout-review prompt

The four-command consumer loop, each verb mapped to the moment it belongs to. Definitions from the consumer ramp (Use APM packages).

Only `apm init` is one-time; the other three recur. Heavier maintenance — `update`, `outdated`, `audit` — is deliberate work saved for Chapter 7, not part of the daily path. And the loop does not grow when `GOVERNANCE` arrives: "Consumer commands are unchanged when policy is enforced — you'll just see more `[x]` blocks at install time when something is denied" (Use APM packages). The same four verbs survive Chapters 8–9 intact.

Restore vs. add: one command, two jobs

The single command `apm install` does two conceptually different things depending on whether you hand it an argument. "With no arguments it installs everything from `apm.yml`. With one or more `PACKAGE_REF` arguments it adds those packages to `apm.yml` … and installs only what was added" (apm install). Bare install is a **restore** — it replays the manifest and lockfile and deploys the same bytes. `apm install <pkg>` is an **add** — it resolves a new dependency, writes it into `dependencies.apm`, and re-locks.

These are the two halves of team dependency work. *Add* is how the manifest grows — a reviewable change to project state, exactly like `npm install <pkg>` editing `package.json`. *Restore* is how everyone else catches up without thinking, and it is where the whole book's onboarding promise lands: **`git clone` + `apm install`** replaces the wiki page, the Slack thread, and the "copy my prompt files" dance from Chapter 1. Crucially, restore *reproduces*; it does not *upgrade*. "To refresh dependencies to their latest matching versions or refs, use `apm update`" (Install packages) — that consent-gated move is Chapter 7. Conflating "restore" with "update" is the one error this chapter most wants to prevent.

• MERIDIAN

Anika — the new joiner whose onboarding in Chapter 1 meant a wiki page, a Slack thread, and borrowed prompt files — joins the checkout team. This time she runs two commands:

```
git clone git@github.com:meridian/meridian-checkout.git
cd meridian-checkout && apm install       # bare install = restore from the lockfile
```

She opens the repo in Cursor; a teammate opens the same commit in Copilot. They see the *same* checkout instruction and the same shared context, byte for byte — no wiki, no Slack, no copied files. The Chapter 1 drift is simply gone. Separately, the team does its two everyday things this chapter: someone runs `apm install <pkg>` to *add* a pinned public dependency, and declares a `review` script — advancing the manifest from v0.1.0 to **v0.2.0**. Add grows the manifest; restore reproduces it. That is the whole loop.

In APM

One install command, two jobs

You already met `apm init` in Chapter 4: it "creates a minimal `apm.yml` … so you can start running `apm install` immediately" (apm init), and you run it once. The rest of this section is the other three verbs, each tied to the

concept it implements: *add* and *restore* materialize the manifest, targeting decides where files land, and `apm run` executes a declared script.

Add: `apm install <pkg>` grows the manifest and re-locks

Meridian's team wants a public package's primitives. One command pins it, resolves it (transitively), deploys it into all three harnesses, and updates both `apm.yml` and the lockfile. This is *materialization* in action — a one-line declaration becomes real files the tools read:

Adding one pinned public dependency to Meridian. The add mutates `apm.yml` and re-locks; `--target` wins the precedence chain. NEEDS NETWORK apm v0.23.1

```
$ apm install microsoft/apm-sample-package#v1.0.0 --target copilot,claude,cursor
  [*] Validating 1 package...
  [+] microsoft/apm-sample-package#v1.0.0
  [*] Updated apm.yml with 1 new package(s)                    # <- add mutates the manifest
  [>] Installing 1 new package...
  [i] Targets: claude, copilot, cursor  (source: --target flag)   # <- flag beat apm.yml targets:
    [+] microsoft/apm-sample-package #v1.0.0 @fb285168
    |-- 2 prompts, 3 agents, 4 commands, 3 rules, 1 skill  (across 3 targets)
    [+] <project root> (local)
    |-- 3 rule(s) adopted -> 3 targets  (files unchanged)        # <- your local instruction preserved
  [!] 1 dependency unpinned: github/awesome-copilot -- add #tag or #sha to prevent drift
  [*] Installed 2 APM dependencies in 56.0s.                   # exit 0
```

The add wrote one pinned line into the manifest — `owner/repo#<ref>`, the deterministic form from Chapter 4:

`apm.yml` after the add — the pinned entry APM inserted into `dependencies.apm`. apm v0.23.1

```
dependencies:
  apm:
    - microsoft/apm-sample-package#v1.0.0   # pinned ref -> deterministic restore
  mcp: []
```

And it wrote the lockfile record — the by-product that seeds REPRODUCIBILITY. Notice the two fields that make restore exact: `resolved_commit` (the precise SHA the tag `v1.0.0` pointed at) and `content_hash` (the fingerprint of what was deployed). You do not author this file; APM generates it. Chapter 6 studies it in full — here, just notice it appear:

The lockfile record install writes for the pinned dependency — exact commit + content hash. A generated artifact; never hand-edited. apm v0.23.1

```
# apm.lock.yaml (generated)
dependencies:
- repo_url: microsoft/apm-sample-package
  resolved_commit: fb2851683be0e0e7711421d518bd8dba23b0b1f6       # the exact SHA it became
  resolved_ref: v1.0.0                                           # the ref you pinned
  content_hash: sha256:744cca54cc8ff7ca90aa1dd621c2f98c6291cd793815afe8518001cc94b8aba9
  package_type: apm_package
```

One pinned line plus one command produced roughly twenty real files across three tools — no hand-copying, pinned by commit and hash for the next person. The manifest edit was *declaration*; this command was *materialization*. (The `[!] 1 dependency unpinned` warning is about the sample's own transitive dependency — the lockfile still pins it by `resolved_commit`, so restore stays deterministic; it is a Chapter 6 concern about future drift, not a failure here.)

Restore: bare `apm install` reproduces byte-for-byte

Re-running `apm install` with no argument is the onboarding path from the Meridian beat. It is *idempotent and cache-served*, not a fresh download: APM replays the pipeline from the content-addressed `apm_modules/` cache and re-emits a byte-identical lockfile.

Bare `apm install` restores from the lockfile — served from cache (~10s vs ~56s cold), and the lockfile is byte-stable. This is the `git clone` + `apm install` promise. apm v0.23.1

```
$ apm install                                          # bare = restore (no package argument)
  [i] Targets: claude, copilot, cursor  (source: apm.yml)  # <- now from the manifest, not a flag
    [+] microsoft/apm-sample-package #v1.0.0 @fb285168 (cached)   # <- served from apm_modules/
    |-- (files unchanged)
    [+] <project root> (local)  |-- 3 rule(s) adopted (files unchanged)
  [*] Installed 1 APM dependency in 10.3s.             # exit 0 (vs 56.0s cold)
# apm.lock.yaml is byte-identical -- generated_at unchanged: 2026-07-02T00:01:57...+00:00
```

That byte-stable lockfile is the proof behind the promise: the same declaration produces the same bytes on any machine. Restore *reproduces*; it never chases newer versions. (A stricter, CI-oriented cousin, `apm install --frozen`, refuses to resolve anything new and fails if manifest and lockfile have drifted — the `npm ci` parallel, developed in Chapter 6.)

Targeting at install time — no implicit "one true harness"

Install deploys into **targets** — the target-vs-harness split from Chapter 3, now doing visible work. Which harnesses receive files is resolved by an explicit precedence chain, not a hardcoded favorite: `--target` > `apm.yml` `targets:` > a configured default > auto-detection of harness directories already present (apm install). In the add above, the `--target` flag won (`source: --target flag`); on the next bare install, the manifest won (`source: apm.yml`). The flag is a **per-invocation override** — it did not write `targets:` into `apm.yml`. Pinning `targets:` in the manifest is the recommended way to make deployment identical across machines. And when there is nothing to target, APM does not guess a harness — it stops and teaches (see the pitfalls below). This is PORTABILITY made mechanical: one declared source set, many native destinations.

Run a declared script: `apm run` (experimental) and `apm preview`

The fourth verb executes a script from the manifest's `scripts:` block. It is stamped **experimental** — "the `run` command surface is marked experimental ... flags and behavior may change before 1.0" (apm run). It is also more than literal shell: for a script like `copilot -p .apm/prompts/checkout-review.prompt.md`, `apm run` *compiles* the `.prompt.md` (frontmatter stripped) and injects the resulting *content* into `copilot -p <content>` — the prompt file is compiled, not passed verbatim. Then it shells out to the named runtime. Critically, **APM does not ship the runtimes**: "the runtime CLI must be on your `PATH`" (Run scripts).

`apm preview` is the safe, offline sibling: it compiles the script's prompt and writes it to `.apm/compiled/` *without invoking any runtime* — zero cost, ideal for docs and CI where you only need to see what would run:

`apm preview review` compiles the prompt and invokes no runtime — offline and zero-cost. apm v0.23.1

```
$ apm preview review                                   # compile the prompt, run nothing
  Original command:  copilot -p .apm/prompts/checkout-review.prompt.md
  Compiled command:  copilot                           # <- the -p <file> is compiled out
  Compiled prompt files:  .apm/compiled/checkout-review.txt   # <- prompt body, frontmatter stripped
  [*] Preview complete! Use 'apm run review' to execute.   # exit 0, no runtime invoked
```

You will run the script for real in the worked example below — where the harness-CLI requirement becomes the load-bearing caveat.

◆ FOR ENGINEERING LEADERS

Onboarding becomes *measurable*. The Chapter 1 ritual — a wiki page, a Slack thread, and copying prompt files from a colleague — had no clear "done" and drifted the moment someone edited a local file. Restore replaces it with a two-step, verifiable outcome: `git clone` then `apm install`, after which every developer holds byte-identical context. Make restore part of your repo bootstrap — the `make setup` target, the dev-container `postCreate`, the first line of the contributor guide — so ● PORTABILITY is the default a new hire gets for free, not a checklist they might skip. "Did onboarding work?" stops being a feeling and becomes "did `apm install` exit `0`?"

When to use / pitfalls

Restore or add? Pick by intent

Both jobs are the same command; the argument decides which. Choose by what you are trying to do, and remember that bare install reproduces — it does not upgrade.

You want to...	Run...	Because
set up a freshly cloned repo	`apm install` (bare)	restore reproduces the pinned set from the lockfile
add a new dependency to the project	`apm install <pkg>#<ref>`	add writes it into `apm.yml` and re-locks — a reviewable diff
re-materialize after editing `apm.yml` by hand	`apm install` (bare)	it replays the manifest so declared and on-disk state match again
move to newer versions	`apm update` (Chapter 7)	restore never upgrades; version moves are deliberate and consent-gated

Which form of `apm install` to reach for, by intent.

⚠ **Pitfall: a no-target install stops with exit `2`**

Because there is no implicit "one true harness," an install with nothing to target does not guess — it fails closed, writes nothing, and teaches. Verified in v0.23.1 with a targetless manifest and no harness marker directories:

```
$ apm install                                        # targetless manifest, no marker dirs
  [x] No harness detected
  APM scanned for harness markers (.claude/, .cursor/, .github/instructions/, ...) but found none.
  Previously APM defaulted to copilot; this is now explicit.   # <- no implicit "one true harness"
  Fix with one of:
    apm install <pkg> --target claude
  Or declare in apm.yml:
    targets:
      - claude
  [!] Install interrupted after 1.6s.                # exit 2 -- nothing written (no AGENTS.md)
```

Fix: declare `targets:` in `apm.yml` (recommended, so every machine matches) or pass `--target` for a one-off. The tree is left as just `apm.yml` + `.apm/` — no `AGENTS.md` fallback in this version. This is ● PORTABILITY refusing to be silently wrong.

⚠Pitfall: `apm run` needs the harness CLI on your `PATH`

APM is the launcher; the runtime does the work. If the CLI a script names is not installed, `apm` still compiles the prompt, then the shell step fails — verified with a script naming a bogus runtime:

```
$ apm run broken     # script: "nonexistent-runtime-xyz -p .apm/prompts/probe.prompt.md"
  [>] Running script: broken
  Compiling prompt...  +- .apm/prompts/probe.prompt.md
  'nonexistent-runtime-xyz' is not recognized as an internal or external command...
  [x] Script execution error: Script execution failed with exit code 1   # exit 1
```

That failure is a **missing harness CLI, not an `apm` bug.** In a clean CI box with no runtime installed, `apm run` fails exactly this way — so gate CI on `apm preview` (offline, zero-cost) instead, and reserve `apm run` for machines where the runtime (`copilot`, `claude`, ...) is actually present. The runtime, not `apm`, incurs the latency and credits.

ⓘCommit the clean authored `apm.yml`

`apm install <pkg>` (add) rewrites and *re-serializes* the manifest — it may normalize list style and drop a blank line. Content is preserved; only formatting changes. Bare `apm install` (restore) leaves the file untouched, so the hand-authored form you commit is the restore-stable one. Commit `apm.yml`, `apm.lock.yaml`, and the deployed harness directories; `apm_modules/` is auto-gitignored and rebuilt from the lockfile on every install (Install packages). And remember: `--target` is a per-invocation override, never persisted — if you want a target on every machine, put it in `targets:`.

Worked example

Meridian, committed as v0.2.0

Two everyday actions carried Meridian from the Chapter 4 manifest (v0.1.0: one local instruction, three targets) to **v0.2.0**: an *add* pinned a public dependency, and a `scripts:` entry declared a review workflow. Here is the clean, committable result — the exact file on disk after a restore, since restore never reflows it:

Meridian's `apm.yml` at v0.2.0 — the restore-stable authored form you commit (one pinned dependency + one declared script). `apm v0.23.1`

```
name: meridian-checkout-agent-context
version: 0.2.0
description: Shared AI-agent context for the Meridian checkout service.
author: Meridian Checkout Team
targets:
  - copilot
  - claude
  - cursor

dependencies:
  apm:
    - microsoft/apm-sample-package#v1.0.0     # added this chapter -- pinned, deterministic
  mcp: []
includes: auto
scripts:
  review: "copilot -p .apm/prompts/checkout-review.prompt.md"   # apm run review -> the checkout-review
prompt
```

With the script declared, `apm list` shows it, and `apm run review` executes it — provided the `copilot` runtime is on `PATH`:

`apm list` renders the declared scripts from `apm.yml`. `apm v0.23.1`

```
$ apm list
              Available Scripts
  Script   Command
  review   copilot -p .apm/prompts/checkout-review.prompt.md
```

`apm run review` executing for real — **experimental**, and it needs the `copilot` runtime CLI on `PATH` (the runtime does the work and incurs the cost). `apm v0.23.1`

```
$ apm run review                                          # experimental; runtime CLI must be on PATH
  [>] Running script: review
  Compiling prompt...          +- .apm/prompts/checkout-review.prompt.md
  Executing copilot runtime... Command: copilot   Prompt content: 253 characters   # <- content injected
  ... copilot runs the checkout review ...
  [+] Copilot execution completed successfully (188.68s)
  [*] Script executed successfully!                       # exit 0 (the runtime did -- and billed -- the
work)
```

That is the loop, end to end: the team *added* a dependency and a script to reach v0.2.0, and Anika *restored* the whole thing with `git clone` + `apm install`. Same declaration, same bytes, in Copilot and Cursor alike — PORTABILITY, materialized.

Recap & next

Recap

- **Install materializes the manifest.** `apm.yml` is a recipe; install is the meal — it resolves, gates, scans, deploys, and writes `apm.lock.yaml`. This is where PORTABILITY becomes a fact, and it seeds REPRODUCIBILITY as a by-product.
- **The daily loop is four commands, one habit:** `apm init` once, `apm install <pkg>` to add, bare `apm install` to restore, `apm run <script>` to run. Everything heavier is lifecycle (Chapter 7).
- **One command, two jobs.** Bare install *restores* (reproduce the pinned set, cache-served, byte-stable); `apm install <pkg>` *adds* (mutate `apm.yml`, re-lock). Restore reproduces — it never upgrades.
- **Targets resolve by an explicit chain** (`--target` > `apm.yml` `targets:` > configured default > auto-detect), `--target` is a per-run override, and a no-target install exits `2` rather than guessing — there is no implicit "one true harness."
- **`apm run` is experimental** and shells out to a runtime CLI you supply on `PATH`; `apm preview` compiles the prompt offline at zero cost. A missing runtime is a harness problem, not an `apm` bug.
- Meridian reached **v0.2.0** (one pinned dependency + one script), and Anika onboarded with `git clone` + `apm install` — the Chapter 1 drift, solved.

Next

Every install quietly wrote `apm.lock.yaml`, and every restore reproduced it byte for byte. Chapter 6 — The Lockfile & Reproducibility opens that file up: what `resolved_commit` and `content_hash` guarantee, how `apm install --frozen` turns restore into a strict CI gate, and why byte-for-byte reproduction is the property the whole supply-chain story rests on.

PART III · REPRODUCIBLE BY LOCKFILE / CHAPTER 6

The Lockfile & Reproducibility

Reproduce an APM setup exactly and explain how the lockfile supports that guarantee.

Objective

After this chapter you can read `apm.lock.yaml` and say exactly what it guarantees — the three levels of identity it records (`resolved_ref` → `resolved_commit` → `content_hash`), why bare `apm install` reproduces the same bytes instead of chasing whatever `main` points at today, and how to turn that guarantee into a fail-closed CI gate with `apm install --frozen`. You will also use `apm lock` to write the contract *without* deploying, and `apm find` to trace any deployed file back to the package that produced it. Chapter 5 made the lockfile appear as a by-product; here it becomes the subject — the moment REPRODUCIBILITY stops being a promise and becomes a mechanism.

Concept/Theory

A manifest resolves; a lockfile freezes

Every install in Chapter 5 quietly wrote `apm.lock.yaml`, and you were told to notice two fields inside it — `resolved_commit` and `content_hash` — without yet studying them. This is the chapter that opens that file up, because it answers the question Chapter 5 deliberately deferred: *why does re-running `apm install` reproduce the same bytes instead of fetching whatever is newest?*

The answer is a distinction. A **manifest declares intent**, and intent is often a moving target: a dependency's ref can be a branch (`main`), a tag (`v1.2.0`), a commit SHA, or a semver range (`^1.2.0`). A branch pointer moves with every upstream push; a range admits whatever tag is newest the moment you run install. So two developers installing the *same* `apm.yml` a week apart — or a laptop and a CI runner installing it the same minute — can resolve *different* bytes. That is the agent-context version of "works on my machine."

The **lockfile records the resolution**: the exact commit each declared dependency resolved to, for the whole transitive graph, plus a content fingerprint of the bytes that landed. Where the manifest says "give me something matching this," the lockfile says "here is exactly what you got." Its very first job, per the spec, is reproducibility: "`apm install --frozen` reinstalls the exact commits recorded here — no resolution, no network drift" (Lockfile specification). Subsequent installs *replay* the lockfile rather than re-resolving — which is why restore is deterministic, and why the manifest, not the lockfile, is the thing you edit by hand.

Three levels of identity

The heart of the chapter is that each lockfile entry pins identity at **three** levels, and keeping them straight is what makes "the same skill" a falsifiable claim rather than a hope:

Level	Field	The question it answers
Intent	`resolved_ref`	*What did you ask for?* — the ref from `apm.yml` (`v1.0.0`, `main`, a SHA)
Resolution	`resolved_commit`	*What did it resolve to?* — the exact 40-char commit SHA. **The pin.**
Content	`content_hash`	*What are the bytes, exactly?* — a SHA-256 fingerprint of the source tree

The three identity levels every lockfile entry records, from intent to bytes. Field definitions from the Lockfile specification.

A commit SHA answers *where* the content came from; a content hash answers *what the content is*. Agent context is executable — a prompt is a program for a model — so "the same review prompt" has to mean the same *characters*, not merely the same source pointer. That is why the lockfile records both the pin *and* the hash: "`content_hash` —

SHA-256 of the package file tree — makes ‘same install on every clone’ mean byte-for-byte the same” (The three promises). The hash is also the layer that SECURITY and provenance rest on: if the recorded hash and the on-disk bytes ever diverge, something changed — full stop.

The hash follows the bytes, not the ref

Pinning is by *commit*; integrity is by *content* — and the two can move independently. Meridian's one transitive dependency makes this concrete. In Chapter 5 the direct dependency was pinned (`microsoft/apm-sample-package#v1.0.0`), but it pulled in a skill from `github/awesome-copilot` that resolved **unpinned** — APM even warned `[!] 1 dependency unpinned`. Because `#main` floats, its `resolved_commit` moved between sessions (from `a4aebcd4…` in Chapter 5 to `3169734b…` here). Yet its `content_hash` stayed **byte-identical** (`sha256:9236d06a…`): the ref floated onto a new commit whose bytes were the same, so the fingerprint did not budge. That is content-addressing — and it is also the warning behind the unpinned notice. Pinning freezes the *commit*; if you want the commit frozen too, pin the ref. Deliberately moving to a newer commit is the drift door Chapter 7 opens with `apm update`.

Generated, never hand-edited

`apm.lock.yaml` is a **build product of resolution**, not an authored file. You edit `apm.yml`; APM writes the lockfile. That one rule is what keeps it trustworthy: its entire value is that it is a faithful, mechanical record of what resolution produced, so the review story lives in the *diff*. Change the manifest, regenerate, and the lockfile diff shows exactly which commits and hashes moved as a consequence — a dependency change you review like any other. And reproducibility is not staleness: the lockfile guarantees the *same* bytes on replay; moving to *newer* bytes is a separate, consent-gated act (`apm update`, Chapter 7).

• MERIDIAN

Priya adds a new shared dependency to the checkout project — a branch-pinned package — and pushes the branch. On her laptop everything works: `apm install` resolved the package and updated `apm.lock.yaml` locally. But she commits only the `apm.yml` change and *forgets to commit the regenerated lockfile*. CI runs `apm install --frozen`, and the build **fails**: the manifest now names a package the committed lockfile does not pin, so manifest and lockfile *disagree*. Reproducibility caught the drift instead of shipping it. The fix is *not* to hand-patch the lockfile — Priya runs `apm install` to reconcile, **reviews the lockfile diff in the pull request** (the new package's `resolved_commit` and `content_hash` appear), and commits it. `--frozen` passes again, and the team adopts a rule: the lockfile is the reproducibility contract, reviewed and committed like the code it pins. Note what did *not* happen — no upstream ref “moved”; the break was a manifest that had outrun its own committed lockfile. (Whether to take a *newer* version at all is Chapter 7.)

In APM

Inside `apm.lock.yaml`

Here is the actual lockfile `apm install` wrote for Meridian v0.2.0, annotated field by field. Read it top to bottom: a small header, then a flat `dependencies` list holding *both* direct and transitive packages, then a block for your project's own materialized files. Every dependency pins by `resolved_commit` and fingerprints by `content_hash`.

The lockfile `apm install --target copilot,claude,cursor` generated for Meridian v0.2.0, annotated. Repetitive path/hash lines are elided (…) for readability; every distinct field is shown. Verified on `apm v0.23.1`.

```
lockfile_version: '1'                          # schema version of the lockfile FORMAT itself
generated_at: '2026-07-02T00:53:23...+00:00'  # UTC time the RESOLVED CONTENT last changed; stable across
restores
apm_version: 0.23.1                            # the CLI that wrote this lock (provenance / debugging)
dependencies:                                  # every resolved package: DIRECT and TRANSITIVE, one flat
list
- repo_url: microsoft/apm-sample-package       # the source repo (owner/repo shorthand)
  name: apm-sample-package
  host: github.com
  resolved_ref: v1.0.0                         # INTENT: the ref you pinned in apm.yml
  resolved_commit: fb2851683be0e0e7711421d518bd8dba23b0b1f6   # RESOLUTION: the exact 40-char SHA -- the
pin
  version: 1.0.0                               # the package's declared semver (from its own apm.yml)
  package_type: apm_package                    # kind of package; drives how it deploys
  deployed_files:                              # DEPLOYMENT RECORD: every file this package materialized
  - .github/agents/design-reviewer.agent.md
  - .github/instructions/design-standards.instructions.md
  - .claude/agents/design-reviewer.md
  - .cursor/agents/design-reviewer.md
  - .agents/skills/style-checker/SKILL.md
  # ... 11 more deployed paths across .claude/ .cursor/ .github/ ...
  deployed_file_hashes:                        # CONTENT: SHA-256 of each deployed file -- what `apm audit`
checks
    .github/agents/design-reviewer.agent.md: sha256:616988766e...   # one source primitive ->
    .claude/agents/design-reviewer.md:       sha256:616988766e...  #   identical bytes in every harness
    .agents/skills/style-checker/SKILL.md:   sha256:1142700284...
    # ... 11 more path -> sha256 entries ...
  content_hash: sha256:744cca54cc8ff7ca90aa1dd621c2f98c6291cd793815afe8518001cc94b8aba9  # fingerprint of
the SOURCE tree
- repo_url: github/awesome-copilot             # the TRANSITIVE dep (pulled in by the package above)
  name: awesome-copilot
  host: github.com
  resolved_commit: 3169734bc2fb25d5e092130fc93d24b0dee3ac3a   # FLOATS: unpinned #main -- do not treat as
fixed
  version: unknown                             # unpinned: no tag resolved (hence the [!] warning at
install)
  virtual_path: skills/review-and-refactor     # the sub-path carved from the monorepo (one primitive, not a
whole pkg)
  is_virtual: true                             # a "virtual" package: a slice of a repo
  depth: 2                                      # 1 = direct from your apm.yml; 2 = transitive
  resolved_by: microsoft/apm-sample-package    # PROVENANCE: the parent edge that introduced this dep
  package_type: claude_skill
  deployed_files:
  - .agents/skills/review-and-refactor/SKILL.md
  - .claude/skills/review-and-refactor/SKILL.md
  content_hash: sha256:9236d06a1500089ddb46975b866e9a63478e502afe7095b1980c618678a7c7fe  # IDENTICAL to Ch5
though the commit moved
local_deployed_files:                          # your project's OWN .apm/ sources, materialized (no
dependency involved)
- .github/instructions/meridian-checkout.instructions.md
- .github/prompts/checkout-review.prompt.md
- .claude/rules/meridian-checkout.md
# ... 3 more local paths across .claude/ .cursor/ ...
local_deployed_file_hashes:                    # SHA-256 of each local file (environment-specific: line
endings)
  .github/instructions/meridian-checkout.instructions.md: sha256:bf0a9567...
  # ... 5 more local path -> sha256 entries ...
```

A few things are worth naming. The `deployed_files` / `deployed_file_hashes` pair is the **deployment record** — the map `apm audit` checks against disk; notice the two `design-reviewer` entries share one hash (`616988766e…`), proof that "one source primitive → many harness files" is still *one* set of bytes. The transitive entry adds three provenance fields — `is_virtual` (a slice of a repo, not a whole package), `depth` (`2` = transitive), and `resolved_by` (the edge that pulled it in). The `local_deployed_*` block is your own `.apm/` content; its hashes are environment-specific (line endings), so treat them as local, not as a cross-machine assertion. And there is *no* `mcp:` section, because this project declares `mcp: []` — how MCP servers appear in the lockfile is Chapter 8's story, not something to infer here.

One field deserves a caveat: `generated_at` stamps "the time the resolved content last changed," *not* "when you last ran install." It advances only when resolution changes, and is preserved byte-for-byte across restores — which is exactly what makes restore provably deterministic:

Determinism: two consecutive bare `apm install` runs on the unchanged project leave the lockfile byte-identical — `generated_at` does not even advance. Cache-served, offline. `apm v0.23.1`

```
$ apm install ; apm install            # restore twice, no changes
RUN1  generated_at: 2026-07-02T00:58:03...+00:00   sha=EAD3EE9B...747D
RUN2  generated_at: 2026-07-02T00:58:03...+00:00   sha=EAD3EE9B...747D   # <- identical
BYTE_STABLE = True                     # generated_at frozen; file unchanged
```

That is the headline made mechanical: `git clone` + `apm install` lays down the same bytes on any machine, and re-running install is a byte-stable no-op. Reproducibility, proven each time you restore.

`apm install --frozen` — the fail-closed CI gate

Reproducibility is only *real* if something enforces it. A regular install is forgiving — if `apm.yml` gained a dependency that is not locked yet, it will resolve and lock it for you. That convenience is exactly wrong in CI, where a passing build must prove that *what is committed* is internally consistent. `apm install --frozen` makes the lockfile a **contract**: it "mirrors `npm ci`" and refuses to install when the lockfile is missing or has drifted from the manifest (apm install). Critically, it is a **structural presence check**, not a byte comparator. On an in-sync project it behaves like a normal restore and ends with a tell that names its own boundary:

`apm install --frozen` on the in-sync project: a normal restore that ends with the `--frozen` tell — it verified *presence* and points you at `apm audit` for on-disk content. Offline (cache-served), exit `0`. `apm v0.23.1`

```
$ apm install --frozen
  [i] Targets: claude, copilot, cursor  (source: apm.yml)
    [+] microsoft/apm-sample-package #v1.0.0 @fb285168 (cached)   |-- adopted
    [+] github.com/.../review-and-refactor @3169734b (cached)     |-- (files unchanged)
    [+] <project root> (local)                                    |-- adopted
  [*] Installed 2 APM dependencies in 13.0s.
  [i] Lockfile presence verified. Run 'apm audit' for on-disk content integrity.   # <- the --frozen tell
```

That last line is the load-bearing nuance of the whole chapter. `--frozen` guarantees the *graph is the locked graph* — the lockfile exists and the manifest agrees with it. It does *not* re-hash your deployed files. Here is precisely what does and does not trip it, all verified on v0.23.1:

Situation	`--frozen`	Why
Lockfile missing entirely	**FAIL** (exit `1`)	nothing to reproduce from
A package declared in `apm.yml` is absent from the lock	**FAIL** (exit `1`)	manifest ↔ lockfile disagree — *the real CI break*
Change a pinned ref to another ref at the *same* commit (`#v1.0.0` → `#main`)	PASS	the package is still present; the ref *string* is not what `--frozen` guards
Add a dependency whose repo is already locked (even transitively)	PASS	the repo is present; it is promoted, not "missing"
A floating ref drifts to a *different* commit	PASS (not caught)	structural, not a commit comparator — the `resolved_commit` pin holds this
A deployed file is hand-edited on disk	PASS (not caught)	structural, not content — that is `apm audit`'s job

What `apm install --frozen` catches — and, just as importantly, what it does not.

So the running-example break is *not* "a ref moved upstream." It is a package that is **declared in the manifest but missing from the committed lockfile** — someone added a dependency and did not commit the re-locked file. The docs are explicit about the boundary: "This is a structural check, not a content check — run `apm audit --ci` for hash verification" (apm install). Pair them and never conflate them: `--frozen` = *the graph is the locked graph*; `apm audit` = *the bytes are the locked bytes* (the audit command lands in Chapter 7, and Chapter 8 builds its security guarantees on the same content hash).

`apm lock` — write the contract without deploying

Sometimes you want the lockfile refreshed but you do *not* want to touch your working tree — a PR check that only asks "does this manifest resolve cleanly?", or a headless job that regenerates the lock, or an SBOM export. `apm lock` is that tool: it "resolves dependencies and writes `apm.lock.yaml` without deploying files" (apm lock). We proved the no-deploy guarantee by deleting the lockfile *and* one already-deployed file as a canary, then running `apm lock`:

`apm lock` resolves the graph and writes the lockfile but **deploys nothing** — a deleted deployed file (the canary) is *not* recreated, and the lock it writes omits the deployment record. NEEDS NETWORK apm v0.23.1

```
$ del apm.lock.yaml
$ del .github/instructions/design-standards.instructions.md    # canary: an already-deployed file
$ apm lock
  [+] microsoft/apm-sample-package #v1.0.0 @fb285168
  [+] github.com/github/awesome-copilot/skills/review-and-refactor #default @3169734b
  [+] Lockfile written to apm.lock.yaml
# lockfile regenerated? True     canary recreated? False        # <- resolved + locked, but NOT deployed
# the new lock has NO deployed_files / local_deployed_files     # <- resolution-only
```

That last comment is the catch: an `apm lock` file is **resolution-only**. It records `resolved_commit` / `resolved_ref` / `content_hash` per dependency but *omits* `deployed_files`, `deployed_file_hashes`, and the `local_deployed_*` block — because it never deployed. It is perfect for validation and `apm lock export` (SBOM/inventory), but it is not the same artifact `apm install` commits: a later install will add the deployment record and change the file. Use `apm lock` to *compute* the contract; use `apm install` to compute *and materialize* it.

`apm find` — trace any file to its origin

Reproducibility is worthless if you cannot answer "*where did this file come from?*" `apm find` closes that loop. Reading straight from `apm.lock.yaml`, it prints the package that owns a deployed file; `--source` appends the resolved coordinate, and `--path` prints the full root-to-target chain, including the transitive edge:

`apm find` traces a deployed file to the package that produced it, straight from the lockfile. `--source` adds the resolved coordinate; `--path` prints the full provenance chain, including the transitive edge. The `awesome-copilot@3169734b` coordinate is unpinned — the tip at capture time; `#main` floats, so your run may differ. Offline. `apm v0.23.1`

```
$ apm find .github/instructions/design-standards.instructions.md
  microsoft/apm-sample-package                          # the owning package

$ apm find .github/instructions/design-standards.instructions.md --source
  microsoft/apm-sample-package  microsoft/apm-sample-package@v1.0.0    # pinned -> @tag

$ apm find .agents/skills/review-and-refactor/SKILL.md --path
  github/awesome-copilot
    apm.yml -> microsoft/apm-sample-package -> github/awesome-copilot  # the transitive chain

$ apm find .agents/skills/review-and-refactor/SKILL.md --source
  github/awesome-copilot  github/awesome-copilot@3169734bc2fb          # unpinned -> @commit

$ apm find .github/instructions/meridian-checkout.instructions.md --source
  .  (workspace)                                        # your own .apm/ source, not a package

$ apm find .github/does-not-exist.md
  [x] '.github/does-not-exist.md' is not tracked by any installed package in apm.lock.yaml.   # exit 1
```

Notice how the coordinate mirrors the pin: a pinned dependency resolves to `pkg@v1.0.0` (the tag), an unpinned transitive one to `pkg@<commit>`, and a local file to `. (workspace)`. Because `apm find` reads the lockfile, it is only ever as honest as the committed lock — one more reason the lockfile is the artifact this whole chapter is about.

◆ FOR ENGINEERING LEADERS

Reproducibility is the bridge between developer experience and compliance, and the lockfile is its load-bearing beam. On the DX side, `git clone` + `apm install` hands every developer byte-identical agent context. On the compliance side, that same committed lockfile is an *audit record* — it states precisely which commits and which bytes produced a given build, and `apm find` can trace any file an agent read back to the dependency edge that introduced it. The stakes are simple: if you cannot say *which agent context produced a change*, you cannot confidently investigate a regression or answer an audit question. Make `apm install --frozen` a required CI check and treat the `apm.lock.yaml` diff as a reviewable dependency change, and "which context shipped?" stops being an archaeology project and becomes a line in the pull request.

When to use / pitfalls

Treat the lockfile as a contract

Four moves read or write that contract; reach for them by intent. Bare restore reproduces; the other three verify, regenerate, or trace.

You want to...	Run...	Because
set up or re-materialize a cloned repo	`apm install` (bare)	restore replays the locked graph byte-for-byte (Chapter 5)
gate CI on reproducibility	`apm install --frozen`	fails closed if `apm.yml` and `apm.lock.yaml` disagree
regenerate/review the lock without touching your tree	`apm lock`	resolves and writes the lock, deploys nothing (PR checks, SBOM)
find where a deployed file came from	`apm find <file>`	traces it to the owning package and its provenance chain
move to newer versions	`apm update` (Chapter 7)	reproducibility is not staleness; version moves are deliberate

Which lockfile-related command to reach for, by intent.

⚠Pitfall: never hand-edit `apm.lock.yaml`

The lockfile is trustworthy *because* it is derived. Hand-patching it to make CI green produces a lockfile that no resolution actually created — a fiction that will mislead the next restore and the next audit. When `--frozen` reports drift, the correct fix is to change the manifest if needed and **regenerate** (`apm install`, or `apm lock` to review before deploying), then commit the resulting diff. Fix the recipe and rebuild the record; do not forge the record.

⚠Pitfall: `--frozen` checks structure, not bytes

`apm install --frozen` proves the manifest and lockfile *agree* and the pins are present. It will happily **pass** a floating ref that drifted to a different commit, or a deployed file someone hand-edited on disk — those are content questions, and content is `apm audit`'s department (Chapter 7, with Chapter 8 building on the same content hash). The CLI even says so on every frozen run: `Lockfile presence verified. Run 'apm audit' for on-disk content integrity.` Run both in CI — `--frozen` for the graph, `apm audit --ci` for the bytes.

ⓘCommit the re-locked file — and review its diff

Because *add* and *update* both rewrite `apm.lock.yaml`, the diff surfaces in the pull request; review it with the same seriousness as a `package-lock.json` change. And remember the other `--frozen` failure mode — if the lockfile is not committed at all, a fresh checkout has nothing to reproduce from:

`apm install --frozen` with no committed lockfile: the gate refuses before any network access — there is nothing to reproduce from. Exit `1`, offline. `apm v0.23.1`

```
$ apm install --frozen                              # apm.lock.yaml never committed / deleted
  [>] Installing dependencies from apm.yml...
  --frozen requires apm.lock.yaml to exist. Run 'apm install' (without --frozen) or 'apm update'
first.
  [i] Tip: run 'apm outdated' to see what changed, then 'apm update'.
  [!] Install interrupted after 0.3s.               # exit 1 -- nothing written
```

Commit `apm.yml`, `apm.lock.yaml`, and the deployed harness directories; `apm_modules/` is auto-gitignored and rebuilt from the lockfile on every install.

Worked example

The CI break, mechanically

Meridian shipped **v0.2.0** in Chapter 5 with one pinned dependency and one `review` script. Now Priya adds a second, branch-pinned dependency. On her laptop `apm install` resolves it and updates `apm.lock.yaml` — but she commits only the manifest edit:

The `apm.yml` change Priya committed — a new branch-pinned dependency (`acme/checkout-guardrails` is a throwaway illustration for this CI-break scenario, not a dependency Meridian adds to its canonical manifest). The regenerated lockfile was *not* committed alongside it. `apm v0.23.1`

```
dependencies:
  apm:
    - microsoft/apm-sample-package#v1.0.0
    - acme/checkout-guardrails#main          # <- added, branch-pinned; lock re-generated locally but NOT
committed
  mcp: []
```

CI checks out that commit — new manifest, old lockfile — and runs the reproducibility gate. It fails closed:

CI runs `apm install --frozen` and fails: the manifest names a package the committed lockfile does not pin. Exit `1` in `0.1s`, with **no network access** — the structural gate fires *before* any clone is attempted. `apm v0.23.1`

```
$ apm install --frozen
  [>] Installing dependencies from apm.yml...
  --frozen: apm.lock.yaml is out of sync with apm.yml.
    - acme/checkout-guardrails is declared in apm.yml but missing from apm.lock.yaml
  [i] Tip: run 'apm outdated' to see what changed, then 'apm update'.
  [!] Install interrupted after 0.1s.                 # <- exit 1, no network, nothing written
```

Read the message literally: the break is a **manifest ↔ lockfile mismatch**, not an upstream ref that moved. The gate never even tried to clone `acme/checkout-guardrails` — it fired in a tenth of a second, before resolution, precisely because the package is *declared but not locked*. (That the illustrative repo does not exist is irrelevant: a real forgotten re-lock produces the identical error, because `--frozen` is checking presence, not fetching bytes.)

The fix is the discipline the whole chapter argues for. Priya does *not* hand-edit the lockfile. She reconciles by running plain `apm install`, which resolves the new dependency, regenerates `apm.lock.yaml` with its pin and fingerprint, and deploys it. The regenerated lockfile's diff is what the pull request reviews — a dependency change, read like any other:

The lockfile diff the reconcile produced — the new dependency arrives with its `resolved_commit` (the pin) and `content_hash` (the fingerprint). Illustrative: the exact SHA and hash depend on the package. `apm v0.23.1`

```
  dependencies:
   - repo_url: microsoft/apm-sample-package
     resolved_ref: v1.0.0
     resolved_commit: fb2851683be0e0e7711421d518bd8dba23b0b1f6
     content_hash: sha256:744cca54...
+  - repo_url: acme/checkout-guardrails              # the new dependency, now pinned
+    resolved_ref: main
+    resolved_commit: <40-char SHA that #main resolved to>
+    content_hash: sha256:<fingerprint of the deployed bytes>
```

She commits the lockfile, and CI re-runs the gate. With manifest and lockfile in agreement again, `--frozen` passes and prints its tell:

With the regenerated lockfile committed, the reproducibility gate is green again. Offline, exit `0`. `apm v0.23.1`

```
$ apm install --frozen
  ...
  [i] Lockfile presence verified. Run 'apm audit' for on-disk content integrity.   # exit 0
```

Nothing exotic happened: a manifest edit outran its own committed lockfile, the gate caught it before touching the network, and a regenerate-review-commit brought the two back into agreement. The team leaves with a rule they now treat as non-negotiable — the lockfile is the reproducibility contract: generated, reviewed in the diff, and committed alongside the manifest that produced it. That is REPRODUCIBILITY, enforced.

Recap & next

Recap

- **The manifest names intent; the lockfile freezes the fact.** `apm.lock.yaml` records the exact resolved graph — direct and transitive — so restore is byte-for-byte. This is where REPRODUCIBILITY becomes a mechanism, not a promise.
- **Three levels of identity:** `resolved_ref` (what you asked for) → `resolved_commit` (the pin) → `content_hash` (the exact bytes). The hash follows the bytes, not the ref — a floated commit with identical content keeps an identical fingerprint.
- **`apm install --frozen` is a structural presence gate,** not a content check and not a ref comparator. It fails when the lockfile is missing or when a package is declared in `apm.yml` but absent from the lock — the real CI break. On-disk byte integrity is `apm audit`'s job, kept deliberately distinct.
- **`apm lock` writes the contract without deploying** (resolution-only; omits the deployment record) — ideal for PR checks and SBOMs. **`apm find`** traces any deployed file to its owning package and provenance chain.
- **Never hand-edit the lockfile.** Fix the manifest, regenerate, review the diff like a dependency change, and commit the re-locked file. Restore is byte-stable — a no-op install even leaves `generated_at` unchanged.
- Meridian's CI break was a **manifest ↔ lockfile mismatch** — a declared package missing from the committed lock — caught offline in `0.1s` and fixed by reconcile-review-commit.

Next

Reproducibility holds the graph *still*; the next question is how to move it *deliberately*. Chapter 7 — Lifecycle opens the intentional-change door named here: `apm outdated` to see what drifted, `apm update` to move pins forward on purpose, and `apm audit` — the content-integrity companion to `--frozen` — to verify that the bytes on disk still match the bytes the lockfile promised.

PART III · REPRODUCIBLE BY LOCKFILE / CHAPTER 7

Lifecycle

Keep APM dependencies current while detecting drift and risk deliberately.

Objective

After this chapter you can run APM's maintenance triad — `apm outdated`, `apm update`, and `apm audit` — and keep the three jobs straight: *report* what has drifted from upstream, *change* versions on purpose, and *verify* the result is intact. You will read an `apm outdated` table, know why a freshly-locked floating ref shows as up-to-date until its branch moves, drive a consent-gated `apm update` (the **only** version-moving verb), and read the `apm.lock.yaml` diff it produces as a reviewable artifact. You will also know precisely what `apm audit` is — a hidden-Unicode and drift-integrity check, *not* an `npm audit`-style vulnerability feed — and when to reach for its `--ci` gate. This is where REPRODUCIBILITY stops meaning merely "frozen" and starts meaning "frozen *and* kept current, deliberately," with a first look at PROVENANCE / SECURITY through the audit check.

Concept/Theory

Reproducible is a floor, not a ceiling

Chapter 6 solved "works on my machine" by freezing the dependency graph: `apm install --frozen` guarantees a clone matches the lockfile byte for byte. But a frozen graph, left alone, quietly rots. Upstream skills gain fixes, prompts get sharper, instructions track new conventions — and a project pinned six months ago is reproducible *and stale*. The failure mode from Chapter 1 simply inverts: instead of silent *drift* you get silent *staleness*. **Reproducibility is a floor, not a ceiling.**

So the obvious next worry is: *if everything is pinned, how does anything ever move forward — safely?* The answer is a deliberate maintenance loop layered on top of the reproducible baseline — a three-beat motion you perform on a cadence, not by accident:

1. **Inspect** what has moved upstream since you locked (`apm outdated`).
2. **Change** on purpose, behind a consent gate (`apm update`).
3. **Verify** the result is intact (`apm audit`).

These are the same motions any team already performs on application dependencies; here they apply to agent context. And they are deliberately *not* part of the daily habit. The consumer ramp draws the line explicitly: after naming the four everyday commands from Chapter 5, it says "everything else is either lifecycle automation (`update`, `outdated`, `audit`) or a workflow extension" (Use APM packages). The broader lifecycle concept sets the rhythm too — you use "`install` and `run` daily, `audit` in CI, and `init` and `compile` rarely" (Lifecycle). This chapter uses *lifecycle* in that narrower, everyday sense: the three maintenance verbs, not the whole `init → install → run → audit` flow.

> ⚠ **Bust this myth up front: `apm audit` is not `npm audit`**
>
> `apm audit` ships **no CVE / vulnerability database** and reports zero "known advisories." Its checks are *content integrity* (hidden Unicode — zero-width characters, bidi overrides, tag characters) and *lockfile / drift consistency*. A clean audit means "no hidden characters and nothing has drifted from the lockfile," **not** "no known vulnerabilities" (apm audit). Treating it as a CVE gate would give a false sense of coverage. This is the single most important misconception this chapter exists to prevent — hold onto it through the whole PROVENANCE / SECURITY story that Chapter 8 continues.

• MERIDIAN

Meridian's checkout context has its **first birthday** with a locked setup. It has been reproducible for a quarter (Chapter 6), and it shows: the public package the team pinned in Chapter 5 pulls in a shared review skill on a moving `#main` branch, and that branch has advanced several commits past the version they locked. Rather than let the pin rot or blindly track the branch, the staff engineer institutes a **monthly agent-context refresh** — a scheduled loop that runs `apm outdated` to see what drifted, `apm update` to move it on purpose, and `apm audit` to confirm the result is intact, then lands the whole thing as a reviewed pull request. Deliberate change, not silent drift. That refresh is this chapter.

In APM

Three verbs, three jobs

APM splits maintenance into three single-purpose commands so that *reporting*, *changing*, and *verifying* never blur together. Bundling "what's new?", "change it," and "is it intact?" into one verb is how package managers historically made updates scary — you never knew what a single command would touch. Three verbs give each step its own, observable blast radius:

Verb	Job	Writes?	Contacts upstream?
`apm outdated`	**report** drift from upstream	No — read-only	Yes
`apm update`	**change** versions — the only mover	Yes — rewrites the lockfile (and manifest)	Yes
`apm audit`	**verify** integrity (content + drift)	No (unless you pass `--strip`, which removes hidden characters)	No — cache-only replay + local scan

The maintenance triad, each verb mapped to its job and what it is allowed to touch. Verified on apm v0.23.1.

Report: `apm outdated` shows drift and changes nothing

`apm outdated` answers "*what has drifted from upstream since we locked?*" without changing anything. It "reads `apm.lock.yaml` and queries each remote to detect staleness" and is explicitly "read-only… does not modify `apm.lock.yaml` or touch `apm_modules/`" (apm outdated). On Meridian's freshly-restored project it finds nothing to do:

`apm outdated` on the freshly-installed Meridian project. The pinned tag has no newer match, and the transitive `#main` skill is locked at the branch's *current* tip, so both read as current. NEEDS NETWORK apm v0.23.1

```
$ apm outdated
  [*] All dependencies are up-to-date           # exit 0
```

That "nothing to do" is the chapter's first non-obvious lesson. A freshly-locked floating ref is **not** outdated, because `apm install` locked `#main` to whatever the branch pointed at *that day* — the lock already equals what the ref resolves to now. Drift is latent in the *ref* and surfaced by *time*, not by how loosely the dependency is pinned. A month later, once the upstream branch has advanced *past* Meridian's locked commit, the same command produces a table:

`apm outdated` once the upstream branch has moved ahead of the locked commit. The pinned tag stays `up-to-date`; the moving `#main` skill is now `outdated`. The tip SHA is illustrative — `#main` floats, so your run resolves to a different commit (the content hash stays the same). NEEDS NETWORK apm v0.23.1

```
$ apm outdated
                                                Dependency Status
┌──────────────────────────────────────────────┬────────────┬────────────┬──────────────┬──────────────┐
│ Package                                      │ Current    │ Latest     │ Status       │ Source       │
├──────────────────────────────────────────────┼────────────┼────────────┼──────────────┼──────────────┤
│ microsoft/apm-sample-package                 │ v1.0.0     │ v1.0.0     │ up-to-date   │ git tags     │
│ github/awesome-copilot/skills/review-and-refactor │ (default)  │ 3169734b   │ outdated     │ git branch   │
└──────────────────────────────────────────────┴────────────┴────────────┴──────────────┴──────────────┘
[!] 1 outdated dependency found                 # exit 0 -- finding drift is not an error
```

Read the columns and the reason for each status becomes clear:

Column	Meaning	Pinned tag dep	Unpinned branch dep
Current	what you have locked	`v1.0.0` (the tag)	`(default)` (the default branch)
Latest	what the ref resolves to *now*	`v1.0.0` (same tag)	`3169734b` (new tip commit)
Status	`up-to-date` / `outdated` / `unknown`	`up-to-date`	`outdated`
Source	how staleness was checked	`git tags`	`git branch`

How `apm outdated` checks each dependency, and why the pinned tag holds while the branch drifts. Verified on apm v0.23.1.

A tag-pinned dep is checked against **git tags** and stays current unless a newer matching tag exists; a branch dep is checked against the **branch tip** and goes `outdated` the moment the branch moves ahead of the locked commit. This is exactly the unpinned hazard Chapter 2 flagged, now shown in its lifecycle consequence. Two more facts worth internalizing: `apm outdated` keys off the *committed lockfile*, not a live `apm.yml` edit — loosening a pin in the manifest without re-installing does not change its report — and its exit code is `0` whether or not anything is outdated. Finding drift is not an error; the summary line is informational (exit `1` occurs only when there is no lockfile at all).

Change: `apm update` is the only version-moving verb

`apm update` re-resolves every dependency in `apm.yml` to "the newest version or Git ref allowed by its constraint, prints a structured plan — added, updated, removed, unchanged — and prompts before touching anything" (apm update). It is "the command that actually changes versions in your project" (Update and refresh). Start with `--dry-run`, the safe preview that "computes and prints the plan ... but never writes and never asks":

`apm update --dry-run` previews the plan and writes nothing. Note the caveat below: the plan over-states movement.
NEEDS NETWORK apm v0.23.1

```
$ apm update --dry-run
  [>] Checking upstream for revision-pin freshness...
  [i] Update plan for apm.yml

    [~] github/awesome-copilot/skills/review-and-refactor
        ref: - (a4aebcd -> -)
        files: .agents/skills/review-and-refactor, .agents/skills/review-and-refactor/SKILL.md,
.claude/skills/review-and-refactor, +1 more

    [~] microsoft/apm-sample-package
        ref: v1.0.0 (fb28516 -> -)
        files: .agents/skills/style-checker, .agents/skills/style-checker/SKILL.md, .claude/agents/design-
reviewer.md, +13 more

    2 updated
    [~] updated

  [i] Dry run: no changes applied. Re-run without --dry-run to update.   # exit 0 -- lock unchanged
```

Do not read that plan too literally. It is a *re-deploy* plan, not a precise "these-commits-will-change" diff: it marks **every** re-resolvable dep `[~] updated` — including the pinned `v1.0.0` dep that will *not* move — and it does not pre-compute the destination SHA, rendering it as `-` (`a4aebcd -> -`). The `2 updated` count over-states real movement. The truth shows only when you run it for real and read the deploy phase (or diff the lockfile). Applied non-interactively as a scheduled job with `--yes`:

`apm update --yes` applies the refresh. The deploy phase tells the true story the plan blurred: the pinned dep re-resolved to the *same* commit; only the moving `#main` skill advanced. NEEDS NETWORK apm v0.23.1

```
$ apm update --yes
  [>] Checking upstream for revision-pin freshness...
  [i] Update plan for apm.yml
    [~] github/awesome-copilot/skills/review-and-refactor    ref: - (a4aebcd -> -)         files: ..., +1
more
    [~] microsoft/apm-sample-package                         ref: v1.0.0 (fb28516 -> -)    files: ..., +13
more
    2 updated
    [~] updated
  [i] Targets: claude, copilot, cursor  (source: apm.yml)
    [+] microsoft/apm-sample-package #v1.0.0 @fb285168 (cached)     # <- pinned: commit UNCHANGED
    |-- 2 prompts adopted -> .github/prompts/
    |-- 3 agents adopted -> 3 targets
    |-- 4 commands integrated -> .claude/commands/, .cursor/commands/
    |-- 3 rule(s) adopted -> 3 targets
    |-- 1 skill(s) integrated -> .agents/skills/, .claude/skills/
    [+] github.com/github/awesome-copilot/skills/review-and-refactor #default @3169734b   # <- MOVED to tip
    |-- (files unchanged)                                    # <- content identical: bytes didn't
change
    [+] <project root> (local)
    |-- 1 prompts adopted -> .github/prompts/
    |-- 2 commands integrated -> .claude/commands/, .cursor/commands/
    |-- 3 rule(s) adopted -> 3 targets
  Updated 2 APM dependencies.                                 # exit 0
```

There is the whole point of the verb, made concrete: the pinned dependency re-resolved to the *same* `@fb285168 (cached)` — a tag pin does not move — while the moving `#main` skill advanced `a4aebcd4 → 3169734b`, the branch tip,

with `(files unchanged)` because the file bytes were identical. Only `apm update` did this; Chapter 5's bare `apm install` would have restored the *old* locked commit even for the floating ref. In an interactive session the prompt "defaults to No," and declining exits cleanly — "no manifest rewrite, no lockfile write, no filesystem changes"; in CI or piped stdin you must pass `--yes` (apm update). One more distinction to file away: `apm update` refreshes *project dependencies*; upgrading the CLI binary itself is a different command, `apm self-update` (Update and refresh).

Verify: `apm audit` checks integrity, not vulnerabilities

`apm audit` confirms deployed context is *intact* with two independent checks: a **content scan** for hidden Unicode in deployed prompt / instruction / skill / rules files, and an **install-replay drift** check that re-materializes from cache into a scratch tree and diffs it against your working tree. This protection "already runs automatically in `apm install`" — `apm audit` is the on-demand re-verification, not a gate you must remember to call to be safe (apm audit). Because it replays from cache and scans local files, it needs no network:

Default `apm audit` on the refreshed project: content scan plus install-replay drift, both clean. Runs offline. `apm v0.23.1`

```
$ apm audit
  [>] Scanning all installed packages...
  [>] Replaying install (cache-only)...
  [+] Replayed 3 package(s)
  [>] Diffing scratch vs working tree...
  [+] No drift detected

  [*] 26 file(s) scanned -- no issues found          # exit 0
```

For CI you want a machine-readable gate, and that is `apm audit --ci`: it adds a nine-check lockfile-consistency table on top of the drift replay. This is the `PROVENANCE / SECURITY` guardrail that makes a refresh *trustworthy*:

`apm audit --ci` — the lockfile-consistency gate. Nine checks, all green. Runs offline. `apm v0.23.1`

```
$ apm audit --ci
  [!] Could not determine org from git remote; enforcement skipped (set policy.fetch_failure_default=block
in apm.yml to fail closed)
  [>] Replaying install (cache-only)...
  [+] Replayed 3 package(s)
  [>] Diffing scratch vs working tree...
  [+] No drift detected

                                        [>] APM Policy Compliance
┌──────────┬──────────────────────────┬──────────────────────────────────────────────────────────────────────────
──┐
│ Status   │ Check                    │ Message
│
├──────────┼──────────────────────────┼──────────────────────────────────────────────────────────────────────────
──┤
│ [+]      │ lockfile-exists          │ Lockfile present
│
│ [+]      │ ref-consistency          │ All dependency refs match lockfile
│
│ [+]      │ deployed-files-present   │ All deployed files present on disk
│
│ [+]      │ no-orphaned-packages     │ No orphaned packages in lockfile
│
│ [+]      │ skill-subset-consistency │ Skill subset selections match lockfile
│
│ [+]      │ config-consistency       │ No MCP configs to check
│
│ [+]      │ content-integrity        │ No critical hidden Unicode or hash drift detected
│
│ [+]      │ includes-consent         │ 'includes:' declared -- local content deployment is explicitly
consented │
│ [+]      │ drift                    │ no drift detected against lockfile
│
└──────────┴──────────────────────────┴──────────────────────────────────────────────────────────────────────────
──┘

  [*] All 9 check(s) passed          # exit 0
```

The choice of form matters because the two modes carry **different exit codes on the same problem** — and that is the difference between an advisory report and a build gate:

Situation	`apm audit` (default)	`apm audit --ci`
Clean	`0`	`0`
Drift only (missing / modified deployed file)	`0` — advisory	`1` — gate fails
Hidden-Unicode finding	`2` — content scan	`1` — via `content-integrity`

The three `apm audit` exit codes, by mode. Default audit is advisory on drift; only `--ci` gates it. Verified on apm v0.23.1.

The benign `Could not determine org from git remote` line appears only because the sample project has no git remote, so *organization*-policy discovery is skipped. In a real repo with a remote, org policy would be fetched and enforced — that fleet story is Chapter 11, not this chapter.

When to use / pitfalls

The division of labor

Six commands touch a locked project, and keeping them straight is most of the skill. Only one moves a version; only one fails a build:

Command	Role	Moves versions?	Fails on drift?
`apm install` (bare)	**restore** locked bytes (Ch 5)	No	No
`apm install --frozen`	**enforce** structural reproduce (Ch 6)	No	Presence gate
`apm outdated`	**report** drift (read-only)	No	No (exit `0`)
`apm update`	**change** — the only mover	**Yes**	n/a
`apm audit`	**verify** content + drift	No	Advisory (exit `0`)
`apm audit --ci`	**gate** lockfile consistency	No	**Yes** (exit `1`)

Every command that touches a locked project, by role. Only `apm update` moves a version; only `apm audit --ci` fails a build. Verified on apm v0.23.1.

The one-line mental model: **install restores · `--frozen` enforces · outdated reports · update changes · audit verifies · `audit --ci` gates.** Note the two integrity checks are *not* the same tool wearing two hats. `apm install --frozen` from Chapter 6 is a *structural* guardrail — it refuses to resolve anything new and fails if the manifest and lockfile have drifted apart before deploying. `apm audit --ci` is an *integrity* guardrail — it replays the deployed tree and runs the nine consistency checks (content hashes, orphaned packages, hidden Unicode, on-disk drift). Use `--frozen` at install time to prove the lock is honored; use `audit --ci` to prove the deployed bytes are intact. Both belong in CI; neither is a CVE feed.

⚠Pitfall: `apm outdated` shows nothing until a branch moves past your lock

On a fresh build, both pinned *and* unpinned dependencies report `up-to-date`. That is not a bug: `apm install` locked the floating ref to the branch's tip at install time, so the lock already equals what the ref resolves to today. A dependency only becomes `outdated` once the branch advances *past* the locked commit; a tag- or SHA-pinned dep stays put indefinitely. Do not expect a loose `#main` pin to light up as outdated the moment you add it — drift lives in the ref and is surfaced by time. If you want a dependency to *stop* drifting, pin it to a tag or full SHA (Manage dependencies); if you want to *surface* drift on a schedule, that is what a periodic `apm outdated` is for.

⚠Pitfall: the `apm update --dry-run` plan over-states movement

The preview marks *every* re-resolvable dependency `[~] updated` — pinned tags included — and renders the destination SHA as `-` because it does not pre-compute it. A `2 updated` line does *not* mean two commits will change. Trust the **lockfile diff** (or the real run's deploy-phase `@<commit>` lines), not the plan's count: in Meridian's refresh the plan claimed two updates, but only one commit actually moved. Use `--dry-run` to confirm *which packages* will be re-resolved and that the run is safe — then read the lock diff to see *what really changed*.

> **⚠Pitfall: default `apm audit` does not gate — use `--ci`**
>
> Bare `apm audit` is **advisory** on drift: it will report a missing or modified deployed file and still exit `0`, because its exit code tracks the hidden-Unicode content scan (which exits `2` on a finding). The same drift exits `1` only under `apm audit --ci`. Wiring bare `apm audit` into a pipeline gives you a check that never fails the build on drift. In CI, use `apm audit --ci` — and remember it verifies integrity, not vulnerabilities.

> **ⓘNever hand-edit the lockfile**
>
> `apm.lock.yaml` is a generated artifact — "regenerated on every install; any manual change is overwritten or ... will trip `apm audit`" (Manage dependencies). The review target is always the *generated* diff from a real `apm update`, never a hand-tweak. And a reminder from Chapter 5: re-running `apm install` (with or without `--force`) restores the locked set and never bumps versions — `--force` only bypasses the security gate for an already-resolved set. Upgrading is always an explicit `apm update`.

Worked example

Meridian's first monthly refresh, as a pull request

The staff engineer's scheduled job runs the triad in order — report, change, verify:

The monthly refresh, three verbs in order. `outdated` and `update` contact upstream; `audit --ci` runs offline. NEEDS NETWORK apm v0.23.1

```
apm outdated            # report: the #main skill drifted past our lock
apm update --yes        # change: move it on purpose, re-lock (the only version-moving verb)
apm audit --ci          # verify: 9 integrity checks pass -> attach as PR evidence
```

The whole reviewable surface of that change is the `apm.lock.yaml` diff. It is plain YAML you can read "to answer 'what version am I actually running?' without trusting the manifest, which may have floating refs" (Manage dependencies). Here it is — and what it does *not* change is as instructive as what it does:

The `apm.lock.yaml` diff a deliberate `apm update` produced — one resolved commit plus the timestamp. The reviewable artifact. apm v0.23.1

```
--- apm.lock.yaml (before update)
+++ apm.lock.yaml (after update)
-generated_at: '2026-07-02T01:17:41.266376+00:00'
+generated_at: '2026-07-02T01:23:19.474392+00:00'   # advanced: update rewrote the resolved content

   - repo_url: github/awesome-copilot                # the transitive #main dep
-  resolved_commit: a4aebcd4bd354b59a6a6c12ab9c5c98a9d9e0276   # old (behind)
+  resolved_commit: 3169734bc2fb25d5e092130fc93d24b0dee3ac3a   # new (branch tip)
```

Field	Before	After	Why
`generated_at`	`…01:17:41…`	`…01:23:19…`	advanced — `update` writes; a pure restore leaves it byte-stable (Ch 6)
`github/awesome-copilot` `resolved_commit`	`a4aebcd4…`	`3169734b…`	the **only** real move — floating `#main` re-resolved to the tip
`github/awesome-copilot` `content_hash`	`sha256:9236d06a…`	`sha256:9236d06a…`	unchanged — the commit moved but the bytes did not (content-addressing, Ch 6)
`microsoft/apm-sample-package` `resolved_commit`	`fb285168…`	`fb285168…`	unchanged — a tag pin does not move under `apm update`

What the refresh changed in the lockfile — and what it deliberately left alone. Verified on apm v0.23.1.

This is the headline of the chapter. An agent-context update is **as reviewable as any dependency bump**. The pull request's review surface is the two files the book has built toward since Chapter 4: the `apm.yml` diff shows *intent* and the `apm.lock.yaml` diff shows *effect*. In this refresh the `apm.yml` is *unchanged* — and that emptiness is itself the evidence that the pins held: the only thing that moved was a transitive floating ref, recorded entirely in the lockfile. A reviewer opens both, sees one `resolved_commit` advance and everything else hold, and reads the attached `apm audit --ci` pass as proof the result is intact. APM frames this mental model directly: for SHA-pinned deps, "think of `apm update` as Dependabot-style review for AI packages" (Manage dependencies). The change is exactly as visible as the team's PR discipline makes it — deliberate change, not silent drift.

◆ FOR ENGINEERING LEADERS

Maintenance that nobody owns does not happen — so a lifecycle loop needs an **owner and a cadence**. APM supplies the tools (`outdated` to surface staleness, `update` to act, `audit` to verify) but deliberately imposes no schedule; the monthly refresh is Meridian's answer, not a product default. Decide who owns update timing — the application team, the platform team, or security enablement — and how stale agent dependencies get reported (a scheduled `apm outdated` is a natural trigger). The payoff is **auditability**: because every deliberate `apm update` lands as an `apm.lock.yaml` diff, "what changed in our agent context, when, and who approved it?" is answered in the same review surface as your application dependencies — and `apm audit --ci` (wired into CI at fleet scale in Chapter 11) gives a machine-checkable ● PROVENANCE / SECURITY signal without pretending to be a CVE feed it is not.

Recap & next

Recap

- **Reproducible is a floor, not a ceiling.** A frozen setup still needs a deliberate maintenance loop — inspect → change → verify — to stay current without giving up ● REPRODUCIBILITY. Staleness is just drift wearing a different mask.
- **Three verbs, three jobs.** `apm outdated` *reports* drift (read-only, exit `0` even when outdated); `apm update` *changes* versions — the **only** version-moving verb, consent-gated; `apm audit` *verifies* integrity.
- **`apm audit` is not `npm audit`.** No CVE database — it checks hidden Unicode and lockfile / drift consistency. Default audit is advisory on drift (exit `0`); `apm audit --ci` gates the nine consistency checks (exit `1`). Distinct from Chapter 6's `apm install --frozen`, which is a *structural* presence gate.
- **Mind the two over-eager reports.** `apm outdated` stays quiet until a branch advances past your lock (pinned deps never light up); the `apm update --dry-run` plan over-states movement — trust the lockfile diff, not the `N updated` count.

- **The lockfile diff is the review artifact.** A deliberate `apm update` lands as a PR: `apm.yml` shows intent, `apm.lock.yaml` shows effect — and here `apm.yml` held while the change lived entirely in the lockfile. `apm audit --ci` is the evidence. Give the loop an owner and a cadence — Meridian chose monthly.

Next

`apm audit` was your first look at the checks that keep agent context trustworthy — and those same hidden-Unicode and content-integrity scans run automatically on *every* install, before anything reaches disk. Chapter 8 — Security by Default opens that install-time pipeline: content-hash pinning, hidden-Unicode blocking, and why transitive MCP servers are held back — the `PROVENANCE / SECURITY` property made mechanical, not optional.

PART IV · SECURE & GOVERNED / CHAPTER 8

Security by Default

Understand and rely on APM's install-time security checks without confusing them with runtime sandboxing.

Objective

After this chapter you can rely on the security checks APM runs on *every* install — and say precisely where they stop. You will read the **hidden-Unicode scan** (a Critical finding exits `1` and blocks deployment; a Warning-only finding exits `2` and ships flagged), use the lockfile's `content_hash` as a **tamper detector** via `apm audit --ci`, explain why a **transitive MCP server is withheld** — a warning, *not* a build break, so the install still succeeds — and the two ways to opt one in, and drive the opt-in **Executable Trust Gate** with `apm approve` / `apm deny`. Above all you will hold the boundary this chapter exists to protect: APM is an *install-time* gate, **not** a runtime sandbox. This is where PROVENANCE / SECURITY stops being a promise and becomes a set of mechanical, always-on defaults.

Concept/Theory

A prompt is a program, so installation is the gate

Chapter 7 introduced `apm audit` as the on-demand integrity tool — hidden-Unicode plus drift, explicitly *not* a CVE feed. This chapter reveals the part that was implied but not stated: **those same checks run automatically on every `apm install`**, before anything reaches disk. You do not have to remember to audit to be safe by default. Understanding *why* the protection has to live at install time is the whole concept.

Start from a claim APM makes about the thing it manages: "*Agent context is executable — a prompt is a program for an LLM. APM treats it that way*" (the three promises). An instruction file steers every generation; a prompt is a runnable procedure; a skill tells the agent *how*; and an MCP server grants it tool *and data access*. Installing an agent package therefore imports behavior the same way importing an npm package imports code — which makes both its **provenance** (where it came from) and its **content** (what the bytes actually say) security-relevant, not stylistic. This is the intuition Chapter 1 named: "*a prompt is, in effect, a program for an LLM, so an unvetted prompt or transitive MCP server is real supply-chain surface.*"

Now the timing. A traditional package manager has a gap you can inspect: "*Between `npm install` and `npm start` there is a gap — time for `npm audit`, code review, and policy checks. Agent configuration has no such gap. The moment a skill, instruction, or prompt file lands in `.github/prompts/` or `.claude/agents/`, any IDE agent watching the filesystem … may already be ingesting it. There is no 'execution step.' File presence IS execution*" (Security model). If there is no install → run gap, a post-install audit is too late — a watching harness may have read the tainted file already. The only place a check can be **preventive** rather than forensic is *before the bytes reach an agent-readable directory*. So APM "*treats package deployment as a pre-deployment gate: scan first, deploy only if clean*" (Security model).

That single idea justifies the whole chapter. An *unvetted prompt* is unreviewed code running against your codebase through an agent; an *undeclared transitive MCP server* is an unaudited capability grant — network, filesystem, tool calls — that arrived as a side effect of installing something else. Every mechanism below is a pre-deploy or on-demand-verify gate, never a runtime monitor. Hold that distinction from the first page:

⚠Bust this myth up front: APM is not a runtime sandbox

APM governs **what reaches disk and whether it conforms** — the install and integrity plane — and hands everything after that to your agent harness. The docs are blunt: "*APM does NOT sandbox MCP servers at runtime, does not do malware analysis on dependency code, does not sign packages, and does not inspect what an agent does once it has read your context*" (Security model), and it has "*no runtime footprint … APM generates files then terminates*". The rule to repeat: "`apm-policy.yml` *governs what gets installed; your agent harness governs what runs. The two planes do not overlap*" (microsoft/apm README). Expecting runtime enforcement from APM is the single most common overclaim, and the one this chapter exists to prevent. Everything here is a *gate*, not a *guard*.

• MERIDIAN

Meridian's checkout context is locked and maintained on a monthly cadence (Chapter 7) — call it `v0.2.0`. This month a teammate finds a genuinely useful internal package, `checkout-review-pack`, that would sharpen the team's review prompt. It looks like harmless markdown. But it *depends on* a small tools package, `checkout-review-tools`, which declares its own MCP server — `local-fetch` — that the team never chose. When the staff engineer adds the pack at `v0.3.0` and runs `apm install`, APM does **not** silently wire up that tool. It prints a `[!]` warning, **withholds** the transitive MCP server, and the install still *succeeds*. The capability that used to be invisible until an audit is now visible at install time — and the fix is a reviewable one. That review, and the `v0.3.1` it produces, is this chapter's worked example. (The private package fetch is `SKIPPED-needs-network`; every mechanism is demonstrated with the verified self-defined-MCP behavior.)

In APM

Four checks, one chokepoint

"Secure by default" is not a slogan; it is four concrete checks that run at the install chokepoint. Three are always on; the fourth is an opt-in gate. APM's own summary of the always-on three: each install "*scans for invisible Unicode that can hijack agent behavior, pins content hashes in the lockfile, and blocks transitive MCP servers unless they are explicitly declared or trusted*" (the three promises).

Check	Job	Always on?	Effect on install
Hidden-Unicode scan	catch invisible instructions in text	Yes	Critical **blocks deploy** (exit `1`); Warning ships flagged
Content-hash verify	detect tampered bytes	Yes	fresh-download mismatch **aborts**; deployed drift surfaces in `audit`
Transitive-MCP withhold	no silent capability grants	Yes	MCP **withheld**, install *succeeds* (exit `0`)
Executable Trust Gate	gate code-bearing primitives	No — opt-in	unapproved executables **parked** pending approval

The four install-time checks, each mapped to the concept it implements and what it does to your install. Verified on apm v0.23.1.

Scan: hidden Unicode can hide instructions in plain sight

LLMs tokenize characters a human never sees on screen, so invisible Unicode is a genuine prompt-injection surface. "*Researchers have found hidden Unicode characters embedded in popular shared rules files. Tag characters (U+E0001–E007F) map 1:1 to invisible ASCII. Bidirectional overrides can reorder visible text. … The Glassworm campaign (2026)*

exploited this mechanism …. LLMs tokenize all of these individually, meaning models process instructions that developers cannot see on screen" (Security model). APM ranks findings by severity, and the severity maps to an exit code:

Severity	Example codepoints	`apm audit` exit	Blocks `apm install` deploy?
Critical	bidi overrides (`U+202A–E`, e.g. RLO `U+202E`), tag chars (`U+E0001–E007F`), Glassworm variation selectors (`U+E0100–E01EF`)	`1`	**Yes**
Warning	zero-width (`U+200B–D`), soft hyphen (`U+00AD`), mid-file BOM	`2` (only if no Critical)	No — ships flagged
Info	unusual whitespace, emoji variation selector (`U+FE0F`), ZWJ inside emoji	`0` (shown with `-v`)	No

Hidden-Unicode severity and the `apm audit` exit code it produces. Verified on apm v0.23.1.

You can point the scan at *any* file with `apm audit --file` — a downloaded rules file, a PR diff, pasted instructions — which isolates the content scan from drift detection. Here is a crafted "review checklist" with a zero-width space on line 4 and a right-to-left override on line 7. The Critical finding decides the exit code:

`apm audit --file` on a file containing a Critical bidi override (`U+202E`) and a Warning zero-width space (`U+200B`). The Critical finding drives the exit code. Runs offline. apm v0.23.1

```
$ apm audit --file checkout-review.md

                          [>] Content Scan Findings
┌──────────┬──────────────────────┬──────────┬────────────┬──────────────────────────┐
│ Severity │ File                 │ Location │ Codepoint  │ Description              │
├──────────┼──────────────────────┼──────────┼────────────┼──────────────────────────┤
│ CRITICAL │ checkout-review.md   │ 7:1      │ U+202E     │ Right-to-left override   │
│ WARNING  │ checkout-review.md   │ 4:1      │ U+200B     │ Zero-width space         │
└──────────┴──────────────────────┴──────────┴────────────┴──────────────────────────┘

[x] 1 critical finding(s) in 1 file(s) -- hidden characters detected
[i]   These characters may embed invisible instructions
[i]   Review file contents, then run 'apm audit --strip' to remove
# exit=1
```

A file whose only finding is a Warning — a lone zero-width space, no Critical — takes a lower severity and a different exit code:

A Warning-only file (zero-width space, no Critical) exits `2`: flagged, not blocked. Runs offline. apm v0.23.1

```
$ apm audit --file warn-only.md
  [>] Content Scan Findings
  WARNING   warn-only.md   3:21   U+200B   Zero-width space
[!] 1 warning(s) in 1 file(s) -- hidden characters detected
[i]   Run 'apm audit --strip' to remove hidden characters
# exit=2
```

`apm audit --strip` removes both Critical and Warning characters in place (preserving emoji and the ZWJ inside emoji sequences). Preview with `--dry-run` first — it writes nothing — then apply, then re-audit clean:

The `--strip` round-trip: dry-run preview (writes nothing), strip in place, re-audit clean. All exit `0`. Runs offline.
`apm v0.23.1`

```
$ apm audit --file checkout-review.md --strip --dry-run
  [>] Dry run -- the following would be removed by --strip:
      checkout-review.md    Critical: 1   Warning: 1   Total: 2
  [i] 1 file(s) would be modified
  [i] Run 'apm audit --strip' to apply
# exit=0    (file UNCHANGED -- U+202E still present)

$ apm audit --file checkout-review.md --strip
  [+]   Cleaned: checkout-review.md
  [*] Cleaned 1 file(s)
# exit=0    (U+202E and U+200B both removed)

$ apm audit --file checkout-review.md
  [*] 1 file(s) scanned -- no issues found
# exit=0
```

This scan already runs *inside* `apm install` (and `apm compile`, `apm unpack`), *before* any integrator copies a file to `.github/`, `.claude/`, or `.cursor/`. A Critical finding **blocks the deploy** — the package stays cached under `apm_modules/owner/package/` so you can inspect it, but nothing reaches an agent-readable directory — and the install exits `1`. Warnings deploy with a flag. The escape hatch is explicit and loud: `apm install --force` "*bypass[es] the security scan's critical-finding block … Use only after independent verification*" (apm install). Two teaching points: `--strip` cleans *deployed* copies but does not touch the source package, so the next install re-materializes the tainted bytes (a durable fix means pinning a clean commit upstream); and the built-in gate is why *you do not have to call* `apm audit` *to be safe by default* — the on-demand command is a power tool, not the protection itself.

> **ⓘ Correction to carry from Chapter 7: hidden-Unicode is not always "exit 2"**
>
> Chapter 7 summarized a hidden-Unicode finding as "exit `2`." That holds only for **Warning-only** files. A **Critical** finding — a bidi override, a tag character, a Glassworm variation selector — exits `1` and blocks the deploy; Warning-only exits `2` and ships; a clean file (or one with Info-only findings, or a `--strip`/`--dry-run` run) exits `0`. In `apm audit --ci` mode the mapping collapses to the usual gate: `0` pass, `1` any failure. Keep the three severities distinct — conflating them either overstates the gate ("any hidden character fails my install") or understates it ("it's only a warning").

Pin: the reproducibility hash doubles as a tamper detector

Chapter 6 introduced the lockfile's `content_hash` (a SHA-256 fingerprint of a dependency's source tree) as a *reproducibility* device: same bytes on every clone. The same field does a second job here. "*On subsequent installs, cached packages … are verified against the lockfile hash. When the on-disk tree no longer matches, APM … re-downloads. If freshly downloaded content still does not match the lockfile record, the install aborts (possible supply-chain tampering)*" (Security model). On the deployed side, `apm audit` verifies each on-disk file — but with a two-tier behavior you must get right. Tamper a deployed file and run a bare audit:

A bare `apm audit` after a deployed instruction file was tampered: drift is reported but **advisory** — exit `0`. Runs offline. apm v0.23.1

```
$ apm audit          # after appending a line to a deployed instruction file
  [>] Scanning all installed packages...
  [>] Replaying install (cache-only)...
  [!] Drift detected: 1 file(s)
  [*] 3 file(s) scanned -- no issues found
    modified (1):
      - .github/instructions/meridian-checkout.instructions.md
    [i] Run 'apm install' to re-sync deployed files with the lockfile.
# exit=0    <- drift alone is ADVISORY in a bare audit
```

Wire `--ci` and the *same* tampered file becomes a hard failure. The `content-integrity` check compares each deployed file's SHA-256 against the lockfile record and prints `expected=` vs `actual=`:

`apm audit --ci` on the same tampered file: the `content-integrity` check fails with `expected=` / `actual=` hashes — exit `1`. This is the tamper gate. The reported check total — here 8 — varies by project (typically 8–9), so it may differ from the count cited in Chapter 7. Runs offline. apm v0.23.1

```
$ apm audit --ci
  | [x] | content-integrity | 1 file(s) with hash drift -- run 'apm install' to restore  |
  | [x] | drift             | drift detected: 1 file(s)                                   |
  content-integrity details:
    - hash-drift: .github/instructions/meridian-checkout.instructions.md
      (dep=<self>, expected=4acba6aa1bee..., actual=bab6d9cb4f14...)
  [x] 2 of 8 check(s) failed
# exit=1
```

The same hash that guarantees byte-for-byte reproducibility catches tampering. Note the division of labor with Chapter 6: `apm install --frozen` is a *structural* guardrail (does the manifest match the lockfile before deploying?) and explicitly *not* a content check — the CLI itself points you to `apm audit` for on-disk integrity. Use `--frozen` to prove the lock is honored; use `audit --ci` to prove the deployed bytes are intact. And one honest limit: a content hash detects *change*, not *intent*. A legitimate upstream update and a malicious tamper look identical to the hash — which is exactly why moving to new bytes is the deliberate, reviewed `apm update` (Chapter 7), never a silent install-time acceptance.

Withhold: a transitive MCP server never auto-arrives

Trust for MCP servers is scoped by **dependency depth**. A server declared by a *direct* dependency — a package you listed in your own `apm.yml` — is auto-trusted, because you chose it. Install a package that declares its own MCP server directly and APM configures it:

A **direct** dependency's self-defined MCP server is auto-trusted — you chose the package. Runs offline (local package). apm v0.23.1

```
$ apm install               # checkout-review-tools is a DIRECT dependency
  [i] Trusting direct dependency MCP 'local-fetch' from 'checkout-review-tools'
  |  [+]  local-fetch -> Copilot, Vscode (configured)
# exit=0    <- direct-dep MCP is auto-trusted
```

Now put that same server one level down — declared by a dependency *of* a dependency — and APM will not wire it up on your behalf. It **withholds** the server, tells you exactly how to opt in, and *the install still succeeds*:

A **transitive** self-defined MCP server is withheld with a `[!]` warning — but the install succeeds (exit `0`). Runs offline (local packages). `apm v0.23.1`

```
$ apm install               # checkout-review-pack -> checkout-review-tools (MCP is transitive)
  [+] checkout-review-pack (local)    |-- (files unchanged)
  [+] checkout-review-tools (local)   |-- (files unchanged)
  [!] Transitive package 'checkout-review-tools' declares self-defined MCP
      server 'local-fetch' (registry: false). Re-declare it in your apm.yml or
      use --trust-transitive-mcp.
  [*] Installed 2 APM dependencies in 0.8s.
# exit=0    <- the MCP is WITHHELD, but the install SUCCEEDS
```

The `[!]` message names two remedies, and they are *not* equivalent. Re-declaring the server in your own `apm.yml` `dependencies.mcp` promotes it to a **direct** dependency — a reviewed, committed, diffable trust decision the whole team sees. Passing `--trust-transitive-mcp` is a **blanket, per-install** opt-in for all transitive self-defined servers in that run, with no manifest record. Prefer re-declaration for anything shared; reserve the flag for trusted, throwaway environments. The flag path:

`--trust-transitive-mcp` opts the withheld server in for this install. Runs offline. `apm v0.23.1`

```
$ apm install --trust-transitive-mcp
  [i] Trusting self-defined MCP server 'local-fetch' from transitive package
      'checkout-review-tools' (--trust-transitive-mcp)
  |  [+]  local-fetch -> Copilot, Vscode (configured)
# exit=0    <- now .vscode/mcp.json IS written
```

⚠Correction: the transitive-MCP block is a *withhold*, not a build break

It is tempting to picture a red `[x] Blocked` line that fails the install. That is not what v0.23.1 does for a self-defined transitive server. The prefix is `[!]` (a warning), the MCP server is **withheld** (no `mcp_servers` in the lockfile, no `.vscode/mcp.json` written), and the install **succeeds — exit `0`**. The story is "*the tool was quarantined and told you how to opt in*," not "*the build broke*." If you wire a bare `apm install` into CI expecting a non-zero exit to catch this, you will be surprised — the signal is the `[!]` line and the missing MCP config, not the exit code. (Governance can turn withholding into a hard failure; that is Chapter 9.)

ⓘScope note: registry-referenced and private-package MCP

The mechanism above is verified with a **self-defined** server (`registry: false`, launched over stdio). A transitive server that resolves from an MCP *registry* (`registry: true`), and any server pulled through a *private* package, require network access and credentials — those exact strings are `SKIPPED-needs-network` in this book. The *behavior* is the same in both cases: direct is trusted, transitive is withheld until you declare or trust it. Only the wire format of the message differs.

Gate: the opt-in Executable Trust Gate

MCP servers are not the only capability a dependency can ship. The **Executable Trust Gate** (added in v0.22) governs the primitives that actually run code: hooks (`.apm/hooks/`), `bin/` executables, self-defined MCP servers, and canvas extensions. Text primitives — skills, agents, instructions — are never gated, because they carry no code-execution risk. The gate is **opt-in**: with no `executables:` block in your `apm.yml` (or a non-empty org policy), it is disabled and executables deploy unconditionally — a backward-compatible default. You can see the disabled state:

With no `executables:` block, the gate is disabled — executables deploy unconditionally. Runs offline. apm v0.23.1

```
$ apm approve --list
  [i] Executable-trust gate disabled -- all executables deploy. Add an
      `executables:` block to apm.yml to enable it.
    checkout-review-tools#0.3.0: mcp[+:gate-disabled]
```

Add an `executables:` block (even an empty `{}`) and the gate turns on. Now an unapproved executable — here the very same self-defined MCP server, from a *direct* dependency — is held behind an interactive prompt that defaults to **No**. Declining parks the package; the install still succeeds:

With the gate enabled, an unapproved executable is gated `[y/N]` (default No). Declining parks it — the install still succeeds (exit `0`). Runs offline. apm v0.23.1

```
$ apm install               # after adding `executables: {}` to apm.yml
  1 package(s) declare executable primitives:
    checkout-review-tools#0.3.0 (direct dependency)
      1 MCP server(s)
    These will execute code on your machine when triggered by
    your IDE or by 'apm run'.
    Trust checkout-review-tools? [y/N]: n
  [i] 1 package(s) left parked. Their executables will not run until trusted.
  [i] Deploy later: apm approve checkout-review-tools (then re-run apm install)
# exit=0    <- declined = parked, not failed
```

Approve when you are ready. `apm approve` writes a committed `executables.allow` entry to `apm.yml` — keyed on the **package name**, not a path (see the pitfall in the next section):

`apm approve` records the trust decision in `apm.yml`; `--list` shows the deciding layer flip from `gate-disabled` to `project-allow`. Runs offline. apm v0.23.1

```
$ apm approve checkout-review-tools
  Approved checkout-review-tools#0.3.0: 1 MCP server(s)
  [i] Updated apm.yml executables block (1 approved).
# exit=0

$ apm approve --list
  [i] 1 package(s) with executables (1 allowed type(s), 0 blocked type(s)).
    checkout-review-tools#0.3.0: mcp[+:project-allow]     # deciding layer: gate-disabled -> project-allow
```

The committed, shareable trust decision `apm approve` writes to `apm.yml`. apm v0.23.1

```
# apm.yml after `apm approve checkout-review-tools`
executables:
  allow:
    checkout-review-tools#0.3.0:
      mcp: true
```

`apm deny` is the inverse (deny wins over allow), and `apm policy explain <pkg>` prints which layer decided a package's fate (apm approve). In CI (non-interactive), unapproved executables are *parked* silently and the remedy is printed — the install does not fail; only a *required-but-untrusted* executable hard-fails.

Two gates, kept separate

The subtlest point in this chapter: the MCP trust model and the Executable Trust Gate are **two independent mechanisms**, and merging them will mislead you. They even overlap on one primitive — a self-defined MCP server sits under *both* — which is exactly why the same `local-fetch` server was auto-trusted as a direct dep earlier (the executable gate was off, so only the depth-based MCP model applied) yet prompted for approval once the gate was enabled.

	MCP trust model	Executable Trust Gate (v0.22)
On by default?	Always on	Opt-in (needs an `executables:` block or org policy)
Keyed on	dependency **depth** (direct vs transitive)	executable **kind** (from any dependency)
Covers	MCP servers	hooks, `bin/`, self-defined MCP, canvas extensions
Direct dependency	auto-trusted	gated (when enabled)
Transitive dependency	withheld (re-declare or `--trust-transitive-mcp`)	gated (when enabled)
Manage with	`--trust-transitive-mcp`, re-declaration	`apm approve` / `apm deny` / `apm policy explain`

The two orthogonal trust gates. The MCP trust model is depth-based and always on; the Executable Trust Gate is kind-based and opt-in. Verified on apm v0.23.1.

When to use / pitfalls

Getting the boundaries right

Every mechanism in this chapter is easy to over- or under-trust. Four corrections keep you honest, and a fifth reminder keeps the whole chapter in its lane.

⚠Pitfall: a bare `apm audit` does not gate — use `--ci`

Bare `apm audit` is **advisory** on drift and hash mismatch: it reports a tampered or missing deployed file and still exits `0`. The same problem exits `1` only under `apm audit --ci`, where the `content-integrity` and `drift` checks fail closed. Wiring bare `apm audit` into a pipeline gives you a check that *never fails the build* on a tampered file. In CI, always use `apm audit --ci` (apm audit) — the fleet-scale wiring is Chapter 11.

⚠Pitfall: `apm approve` keys on the package *name*, not the path

The parked-install remedy prints a hint, but the trust store is keyed by package identity (`owner/repo`), not by a local path. Approving with a filesystem path prints `… not found in installed packages` and silently no-ops (it still exits `0`), so it looks like it worked while nothing changed. Approve with the package name — `apm approve checkout-review-tools` — and confirm with `apm approve --list` that the deciding layer flipped to `project-allow`.

⚠Pitfall: `--strip` cleans copies, `--force` is break-glass

`apm audit --strip` rewrites the *deployed* files; it does not modify the source package in `apm_modules/`, so the next `apm install` re-materializes the tainted bytes. For a durable fix, pin a clean upstream commit. And treat `apm install --force` as what it is — a labeled fire exit that bypasses the Critical-Unicode block. Use it only after you have independently verified the finding is benign, and leave a trace in your PR that you did.

> ⓘ **Keep the two gates — and the two planes — distinct**
>
> The MCP trust model (depth-based, always on) and the Executable Trust Gate (kind-based, opt-in) answer different questions; do not describe one in the other's terms. And remember the myth from the top of the chapter: all of this is the **install and integrity plane**. APM decides whether an MCP server's config reaches your disk; your *harness* decides what that server may do once the agent calls it. The scan catches *known invisible-character classes* — it does not detect plain-text prompt injection, homoglyphs, semantic manipulation, or binary payloads, and a clean scan means "no hidden Unicode and no drift," never "this prompt is benign." Human review of *visible* content, plus harness and endpoint controls, still apply (Security model).

With those in hand, the "when" is simple. Rely on the **built-in install scan** for baseline safety on every install — you do not have to call `apm audit` to be protected against Critical Unicode. Reach for `apm audit` on demand to scan an arbitrary file (a downloaded rules file, a PR diff) or to check deployed integrity locally, and for `apm audit --ci` as the branch-protection gate. Re-declare a transitive MCP server you have reviewed and intend to keep; save `--trust-transitive-mcp` for throwaway environments. And enable the Executable Trust Gate (an `executables:` block, or org policy in Chapter 9) when you want approval — not just visibility — before a dependency's code can run.

Worked example

Meridian reviews a transitive tool into the open

Back to the beat. Meridian's staff engineer adds `checkout-review-pack` at `v0.3.0`. The pack is useful markdown, but it depends on `checkout-review-tools`, which declares the self-defined MCP server `local-fetch`. On install, APM withholds that transitive server — and, crucially, the install *succeeds*:

Meridian adds `checkout-review-pack` (`v0.3.0`). The transitive `local-fetch` MCP server is withheld; the install succeeds (exit `0`). The private package fetch is `SKIPPED-needs-network`; the withhold behavior is the offline-verified mechanism shown above.

`NEEDS NETWORK` `apm v0.23.1`

```
$ apm install               # SKIPPED-needs-network: private meridian-finance/checkout-review-pack
  [+] meridian-finance/checkout-review-pack #v0.3.0   |-- (review prompt adopted)
  [!] Transitive package 'checkout-review-tools' declares self-defined MCP
      server 'local-fetch' (registry: false). Re-declare it in your apm.yml or
      use --trust-transitive-mcp.
  [*] Installed 2 APM dependencies.
# exit=0    <- the review pack installed; the transitive MCP server was withheld
```

`SKIPPED-needs-network`: the `meridian-finance/checkout-review-pack` package is private, so cloning it needs a host token and is skipped for deterministic verification. The `[!]` withhold message, the exit code, and the remedies are reproduced offline with a local self-defined-MCP package (the "Withhold" figures above), so the mechanism Meridian relies on is verified even though this exact fetch is not.

This is the moment Chapter 1's open question — "*which MCP servers are installed, and did each come from an approved source?*" — gets an answer. The capability is *visible* at install time instead of surfacing in a later audit. The team runs a short security review of `local-fetch`, decides it is legitimate, and takes the **reviewable** path rather than a blanket `--trust-transitive-mcp`: they re-declare the server in their own `apm.yml`, promoting it to a direct dependency. That ships as `v0.3.1`:

`v0.3.1`: the reviewed MCP server re-declared in `dependencies.mcp`, promoting it to a committed direct dependency. The trust boundary is now a diff. This entry's shape mirrors the verified lockfile `mcp_configs` record; the re-declaration itself was not exercised offline. `apm v0.23.1`

```
# apm.yml -- Meridian checkout context, v0.3.1
dependencies:
  apm:
    - meridian-finance/checkout-review-pack#v0.3.1     # the review pack (private)
  mcp:
    # Re-declared after security review: promotes the transitive server to a
    # reviewed, committed DIRECT dependency. The trust decision is now in the diff.
    - name: local-fetch
      registry: false
      transport: stdio
      command: npx
      args:
        - -y
        - '@modelcontextprotocol/server-fetch'
```

Because `local-fetch` is now a *direct* dependency, the next install auto-trusts and configures it — exactly the direct-dependency baseline verified earlier (`[i] Trusting direct dependency MCP 'local-fetch'…`). The difference from `--trust-transitive-mcp` is the whole point: a teammate opening the `v0.3.1` pull request sees one added `mcp:` entry and knows precisely which capability was granted, by whom, and after what review. The alternative ending is equally valid and worth naming: if the review had gone the other way, the right move is simply to *not* grant the capability — drop the dependency, and `local-fetch` stays withheld. Either way, an invisible transitive capability became an explicit, auditable decision at install time.

◆ FOR ENGINEERING LEADERS

Security by default changes the rollout conversation: developers keep moving quickly, but risky content is checked *before* it becomes local tool access, not after an incident. The reframe is that a transitive MCP server is an **ungoverned capability grant** — the kind of thing that used to stay invisible until a review or an audit surfaced it — and APM now makes that grant an explicit, reviewable decision at install time (a one-line `apm.yml` diff, in Meridian's case). Position it accurately in your control stack: APM is a **pre-deploy gate for agent packages** — controlled sources, locked versions, byte-level integrity, and an install-time capability gate — which is a real, auditable control, but one that *composes with* harness runtime governance and endpoint tooling rather than replacing them. Leaders who understand the boundary buy the right complements; leaders who don't develop a false sense of coverage. What this chapter gives you as a built-in default, Chapter 9 lets you make organizational policy.

Recap & next

Recap

- **The trust boundary is install time.** Agent context is executable — a prompt is a program for an LLM — and there is no install→run gap, so *file presence is execution*. APM's protection is therefore a **pre-deploy gate**: scan first, deploy only if clean.
- **Hidden-Unicode scanning, by severity.** Critical (bidi overrides, tag chars, Glassworm variation selectors) exits `1` and blocks deploy; Warning-only exits `2` and ships flagged; clean/info exits `0`. `apm audit --strip` cleans deployed copies (not the source); `apm install --force` is break-glass.
- **The content hash catches tampering.** Bare `apm audit` reports drift as *advisory* (exit `0`); `apm audit --ci` is the tamper gate (exit `1`, `content-integrity` with `expected=`/`actual=`). Distinct from Chapter 6's structural `--frozen`.

- **Transitive MCP is withheld, not blocked.** Direct-dep MCP is auto-trusted; a transitive self-defined server is withheld with a `[!]` warning while the install *succeeds* (exit `0`). Opt in by **re-declaring** it (reviewed, committed) or with `--trust-transitive-mcp` (blanket, per-install).
- **Two orthogonal gates, one plane.** The MCP trust model (depth-based, always on) and the opt-in Executable Trust Gate (`apm approve`/`apm deny`, kind-based) are separate mechanisms — and all of it is the install/integrity plane. APM is *not* a runtime sandbox; the harness governs what runs.

Next

These are the built-in, always-on defaults every install applies with no configuration. The natural next question is organizational: *which sources, scopes, and primitives are allowed here* — and how do you turn a withhold into a hard block, or a warning into policy? Chapter 9 — Governance & Policy adds the configurable layer on top of this chapter's defaults: an `apm-policy.yml` that enforces GOVERNANCE at install time, with tighten-only inheritance and a warn→block rollout — still install-time, never runtime.

PART IV · SECURE & GOVERNED / CHAPTER 9

Governance & Policy

Write and pilot an apm-policy.yml that enforces agent-package rules at install time.

Objective

After this chapter you can author and pilot an `apm-policy.yml` — the organization's allow/deny contract for agent packages — and enforce it at install time. You will keep GOVERNANCE ("is this *allowed here*?") crisply separate from the PROVENANCE / SECURITY checks of Chapter 8 ("is this *safe*?"), write the **verified** policy schema (including the one field the running example got wrong — target rules live under `compilation.target.allow`, *not* a top-level `targets:` key), and drive the `warn` → `block` rollout that turns a rule on without breaking every repo on day one. You will run `apm audit --ci --policy` and read its two exit codes, inspect the effective policy with `apm policy status`, and reason about tighten-only *enterprise* → *org* → *repo* inheritance. One honesty note up front: at `apm v0.23.1` the policy *engine* is **early preview** — schema, discovery, and inheritance ship today, but enforcement semantics may shift between minor versions, so pin the CLI before you lean on it as a production gate (Policy reference).

Concept/Theory

Two questions at the same gate

Chapter 8 made every install *safe by construction*: hidden-Unicode scanning, content-hash pinning, and transitive-MCP blocking all fire **before any file reaches disk.** Those checks ship with the CLI and answer a question with a *universal* answer — a bidi-override character is dangerous everywhere; a tampered hash is invalid everywhere. But a second question has **no universal answer**: *may our developers install `some-org/some-skill`? is the GitHub MCP server approved? which compilation targets do we support?* That depends entirely on *your* organization's risk posture, contracts, and compliance regime.

That second question is GOVERNANCE — APM's third promise, "governed by policy" (the three promises). Where security is **intrinsic** (built into the tool, non-negotiable), governance is **declared**: an org authors an allow/deny contract in `apm-policy.yml` and APM enforces it at the *same* pre-disk gate as the security checks. Same seam, two different authorities. Crucially, policy "does not scan code semantics or behave like an antivirus. It enforces *declarations* against an allow/deny list before APM writes any file" (Policy files).

	SECURITY (Chapter 8)	GOVERNANCE (this chapter)
Question	Is this artifact safe by construction?	Is this artifact allowed *here*?
Source of authority	Intrinsic — ships in the CLI	Declared — the org authors `apm-policy.yml`
Varies by org?	No — same everywhere	Yes — each org's own contract
Where enforced	Install gate (pre-disk)	Install gate *and* `apm audit --ci`

Security and governance answer different questions at the same install-time gate. Verified against apm v0.23.1.

The two verdicts are **independent, and both must pass.** A package can be perfectly safe (clean scan, valid hash) yet *denied* because it comes from an un-vetted source; an *allowed* source still runs through the security scan. So beware the twin misconception — "governance is just more security," and its inverse, "if it's allowed by policy it must be safe." Neither holds. Policy is not an antivirus, and an allow-list is not a safety certificate.

Policy is deliberately **visible to developers yet authoritative for the org.** It fires on a developer's own local `apm install` and prints an inline violation with a remediation hint — but its source of truth is *not* the developer's checkout. As Chapter 1 foreshadowed, `apm-policy.yml` is an **org-remote artifact**: it lives in your org's policy repo

(protected by CODEOWNERS and branch protection) and is auto-discovered from your project's git remote, not committed as a per-repo default (Policy files). The single distinction to protect above all others is the one the microsoft/apm README draws: `apm-policy.yml` **governs what gets *installed*; your agent harness governs what *runs*. The two planes do not overlap.** Governance is an install-time gate, not a runtime sandbox — the schema has no fields for runtime permissions or agent behavior.

In APM

> **ⓘ Version note: the policy engine is early preview**
>
> At `apm v0.23.1` the policy *engine* is flagged **early preview**: "schema, inheritance, and discovery ship today; enforcement semantics may change between minor versions... Pin to a specific APM version before relying on it as a production gate" (Policy reference). The lockfile-based baseline of `apm audit --ci` from Chapter 7 is stable; the `apm-policy.yml` *enforcement* layer this chapter adds is what carries the preview label. Every field and exit code below was verified empirically against v0.23.1; re-confirm on any CLI bump.

The policy file: `apm-policy.yml`

A policy is a single YAML file whose top-level keys each govern one facet of *what may be installed* — dependency sources, MCP servers and transports, compilation targets, manifest shape, unmanaged files. There are no runtime keys. Here is the verified schema, trimmed to the sections you will actually reach for:

The `apm-policy.yml` schema, verified against apm v0.23.1. The single top-level `enforcement` dial governs every rule; target rules live under `compilation.target.allow`. `apm v0.23.1`

```
name: Example Org Policy
enforcement: warn            # off | warn | block  -- the single top-level dial

dependencies:
  allow:                     # null = no opinion | [] = allow NOTHING | [globs] = only these
    - microsoft/**
  deny:                      # deny always beats allow
    - sketchy-org/**
  require: []                # packages every repo must declare (supports "#version" pins)
  require_pinned_constraint: true   # ban unbounded refs (bare branch / wildcard / open range)

mcp:
  allow: null
  deny: []
  transport:
    allow: [http, stdio]     # subset of: stdio | sse | http | streamable-http
  self_defined: warn         # allow | warn | deny  (inline MCPs declared in apm.yml)

compilation:                 # target rules live HERE -- NOT a top-level `targets:` key
  target:
    allow: [copilot, claude, cursor]

manifest:
  required_fields: [name, version, description]
  scripts: allow             # allow | deny

unmanaged_files:
  action: warn               # ignore | warn | deny
```

The confirmed, enforced top-level keys at v0.23.1 are: `name`, `version`, `extends`, `enforcement`, `fetch_failure`, `cache`, `dependencies`, `mcp`, `compilation`, `manifest`, `unmanaged_files`, `registry_source`, `security`, and

`executables`. One allow-list rule is worth memorizing because it is easy to get backwards: a list left as `null` means "*no opinion*" (allow anything not denied); an **empty** list `[]` means "*allow nothing*"; a populated list means "*only these*." And `deny` is always evaluated first — it beats `allow`.

> **⚠Schema trap: a top-level `targets:` is silently ignored — use `compilation.target.allow`**
>
> APM **silently drops unknown top-level keys** — no error, no warning. The most dangerous instance: writing target rules as a top-level `targets: {allow: […]}` block (a natural guess, and the shape an early running-example draft used). It parses cleanly and enforces *nothing*. The real key is `compilation.target.allow`. The failure is invisible unless you inspect `apm policy status`, where the rule count exposes it:
>
> ```
> # WRONG -- top-level targets:  { allow: [copilot, claude, cursor] }
> "compilation_targets_allowed": -1        # -1 = no rule registered (silently ignored)
>
> # RIGHT -- compilation: { target: { allow: [copilot, claude, cursor] } }
> "compilation_targets_allowed": 3         # rule registered
> ```
>
> If you ever see a top-level `targets:` in a policy during review, it is doing **nothing**. Every policy example in this book uses `compilation.target.allow`.

One gate, two enforcement points

The same policy file is evaluated at two places that share one rule set (Policy files):

- **The install-time preflight gate.** `apm install` resolves the dependency tree, runs the policy gate against the resolved set, *then* writes files. A blocking violation halts the install with a non-zero exit and **nothing reaches disk.** This protects a developer running `apm install` locally — they cannot accidentally deploy a denied package even without CI. There is **no `--policy` flag on `apm install`**; discovery is automatic from the git remote.
- **The CI audit gate.** `apm audit --ci --policy org` runs the *same* policy checks **plus** the baseline lockfile/integrity checks from Chapter 7, and emits SARIF for GitHub Code Scanning. This is the check you wire into branch protection — the authoritative gate for merges, and the only enforcer of the audit-only rules (required fields, scripts, unmanaged files) that the install gate does not check.

This is where the Chapter 8 thread ties back: when an APM package pulls in its *own* MCP dependencies, those transitive servers are resolved and then passed through a **second policy pass**, so no transitive MCP server reaches your runtime config without clearing the same `mcp.allow` / `mcp.deny` / `mcp.transport` rules as a direct one. Do *not* confuse that with the `mcp.trust_transitive` policy field — at v0.23.1 that field **parses but is not enforced**; the actual transitive-MCP gate is the `--trust-transitive-mcp` CLI flag (default deny) you met in Chapter 8 (Policy files).

The `enforcement` dial: `warn` and `block`

The top-level `enforcement` field has three modes — `off`, `warn`, `block` — and it is a **single, uniform dial**: every rule runs under the same mode. (Only two sub-fields carry their own `allow|warn|deny` sense — `mcp.self_defined` and `unmanaged_files.action` — but their pass/fail still surfaces through the top-level dial.) The difference between the two modes you will use is entirely in the **exit code on the same violation**, verified on real violating projects:

Mode	Every check runs?	Violation shown as	Summary	Exit code
`warn`	Yes	`[+]` … `(enforcement: warn)`	`All N check(s) passed`	`0` — measures, does not gate
`block`	Yes	failing row	`1 of N check(s) failed`	`1` — fails closed

The `enforcement` dial, same violation, two modes. Run with `apm audit --ci --policy`. Verified on apm v0.23.1.

That single-column difference — exit `0` vs exit `1` on an identical finding — is the whole mechanism behind the rollout you will run in the worked example.

Inspecting policy: `apm policy status` and `explain`

At v0.23.1, `apm policy` has exactly two subcommands (there is no `check`, `diagnose`, or `show`). `apm policy status` reports the effective posture — discovery outcome, cache, and the *merged* rule counts — and accepts `--policy-source` (`org` | `owner/repo` | `https://…` | a **local path**) so you can preview a file before it lands. It **always exits 0** unless you add `--check`, which turns an unreachable or misconfigured org policy into a non-zero CI pre-flight. The other subcommand, `apm policy explain <package>`, is the *executable-trust* surface from Chapter 8 (the `apm approve`/`deny` decision) — not the `apm-policy.yml` gate, so it is not re-taught here.

Tighten-only inheritance: enterprise → org → repo

Policies compose along a chain — canonically an **enterprise hub → org policy → repo override** — declared with `extends:` (max depth 5; cross-host `extends:` is refused as a credential-leak mitigation). The merge is **tighten-only**: "children can only tighten rules, never relax them… a repo can be more restrictive than the org, but cannot widen what the org has allowed" (Policy files). Because the org policy is discovered from the git remote, the org level is the default authority; a lower level is understood as a stricter *child*, never a replacement. In plain English:

Field kind	Merge rule	What a child can do
allow lists (`dependencies.allow`, `mcp.allow`, `compilation.target.allow`…)	**intersection**	Narrow the list — never widen it
deny lists (`dependencies.deny`, `mcp.deny`)	**union**	Add denies — never remove one
`enforcement`	**max** (`off < warn < block`)	Escalate `warn → block` — never the reverse
`require_pinned_constraint`	logical **OR**	Turn it on — never off if a parent set it
`max_depth`, `cache.ttl`	**min**	Lower the cap — never raise it

How the tighten-only merge composes each field kind. The effective policy is always at least as strict as every level above it. From the policy reference; verified rule shapes at apm v0.23.1.

The consequence a security lead cares about: a repo **cannot** override the org policy to unblock itself. A child that tries to drop `block` back to `warn`, or re-allow a denied source, is rejected. Exceptions therefore flow *upward*, not downward — to exempt a repo you relax the rule *at the parent level* (narrow a `deny` glob, or add the package to `dependencies.allow`), documented in the policy file itself. APM has no first-class waiver field; that file is your audit log. Discovery reinforces the org-remote model from Chapter 1: APM resolves the org from the project's git remote and searches candidate repos in order — `.github`, then `.apm`, then `_apm` — first found wins, cached about an hour. Rolling that baseline across product groups is the fleet story of Chapter 11.

When to use / pitfalls

Roll out by measurement, not by switch

A new policy rule is **retroactive**: the next `apm install` or CI audit in every consuming repo runs it against dependencies that were legal yesterday. Flip straight to `enforcement: block` and every repo that already has a violation fails on its next run — you spend the day rolling back instead of rolling out (Policy pilot). So `warn` is not a lesser mode you skip — it is the **measurement phase** that makes `block` safe. The recommended sequence is four steps:

1. **Author in `warn`** in `<org>/.github/apm-policy.yml`. Nothing breaks; every violation is reported and every install still succeeds.
2. **Read the telemetry** — the warnings in `apm install` output and, canonically, the SARIF that `apm audit --ci --policy org` feeds into Code Scanning — to build a fleet-wide violation list.
3. **Remediate the top offenders** — upgrade or replace the offending dependency (the default fix), or grant an exception by relaxing the rule at the parent level.
4. **Flip to `block`** once the count is zero (or the survivors are documented exceptions). To block *one* rule while others still warn, **stage rules** (clean one, then flip the file) or **split via `extends:`** so a strict child escalates for its scope only — there is no per-rule `enforcement` knob.

> **⚠ Pitfall: `warn`-mode audit exits `0` — it does *not* gate CI**
>
> Under `enforcement: warn`, `apm audit --ci` rewrites violations to *passed* so the exit code stays `0` and the check goes **green**. The visibility lives in the SARIF and the Code Scanning UI — *not* in the red/green check. Do not read "CI is green" as "no violations" during the warn phase: watch Code Scanning, and only `block` actually fails the build (exit `1`). This is exactly why the rollout above treats `warn` as a measurement window, not an on switch.

> **⚠ Pitfall: a local-file `extends:` does not merge a parent**
>
> At v0.23.1, `extends:` only merges a parent fetched as `org`, `owner/repo`, or `https://…`. Point it at a **local path** and `apm policy status` will *record* the reference in the extends chain but the parent's rules are **not merged** — the effective policy is the child's rules alone (verified: the child's `deny` count stayed at its own single entry instead of unioning with the parent's). So you cannot prototype inheritance with two files on disk; real multi-level merge needs a remote parent, which is why the inheritance demos in this chapter are marked `NEEDS NETWORK`. Treat a repo-local policy as "*author and test here, then publish to `<org>/.github`*," not as a local child of a local org.

> **⚠ Pitfall: don't audit-test `compilation.target`, and don't trust `mcp.trust_transitive`**
>
> Two rules mislead in a local demo. The `compilation-target` check needs a *resolved* target; on a static manifest that only lists candidate `targets:` it reports "No compilation target set in manifest" and **passes** — it bites at `apm install`/`apm compile` time, not at audit time. So `compilation.target.allow` is the *correct* key (see the schema trap) but a weak choice for an audit-gate demo; lead with `dependencies.*`, `mcp.*`, or `require_pinned_constraint`. And `mcp.trust_transitive` **parses but is not enforced** — the real transitive-MCP gate is the `--trust-transitive-mcp` flag from Chapter 8. Do not claim the policy field blocks transitive MCP.

Two smaller habits round out the "when to use": wire `apm audit --ci --policy org` (not bare `apm audit`) into branch protection — it is the authoritative gate — and reach for `apm policy status --check` only when you want CI to fail because the org policy is *unreachable or misconfigured*, since plain `apm policy status` never fails. Use `apm audit --ci --policy` to gate on rule *violations*; use `--check` to gate on policy *reachability*. The bypass contract is honest but loud: `apm install --no-policy` and `APM_POLICY_DISABLE=1` skip the *auto-discovered* org policy for a single, logged invocation — but an **explicit `--policy <source>` overrides the bypass** and still enforces, and the baseline lockfile checks are never bypassable.

◆ FOR ENGINEERING LEADERS

Turning on governance is **change management, not a config toggle**. The instinct to flip `enforcement: block` on day one is the fastest way to break every repo at once and burn the org's trust in the whole program. Sequence it instead: ship in `warn`, *measure* the fleet-wide violations through Code Scanning for a sprint or two, have teams *fix the top offenders*, then `block` the highest-risk rules **narrowly** — via staged rules or a strict child policy — while the rest of the org keeps warning. Governance that earns trust by making a problem visible before it becomes blocking will stick; governance that fails everyone's build on a Tuesday will be disabled by Wednesday. Decide three things explicitly: *who owns the org policy repo* (protected by CODEOWNERS/branch protection), *which rules are high-risk enough to block first*, and *how exceptions are granted* — upward, at the parent, in a file that doubles as your audit log. And record the pinned CLI version: at v0.23.1 the engine is early preview.

Worked example

• MERIDIAN

Meridian's **platform team** owns the security-review thread that Chapter 8 opened. Catching one sketchy transitive MCP server in one install was good; encoding the org's rules so *every* repo inherits them is the goal. They author a pilot `apm-policy.yml` for the `meridian-checkout` repo in `enforcement: warn`: allow only `microsoft/**`, `github/awesome-copilot/**`, and `meridian-finance/**` sources; deny the sketchy org; require pinned dependencies; permit only the reviewed GitHub MCP server over `http`/`stdio`; and restrict compilation targets to `copilot`, `claude`, `cursor`. For two sprints they watch the warnings roll in and fix the top offenders — then flip the highest-risk rules to `block`.

Here is the pilot policy the team ships. Note the corrected target rule under `compilation.target.allow` — the one field an earlier draft got wrong. Its `require:` rule names `meridian-finance/meridian-standards`, the org's shared standards package that Meridian actually authors and publishes in Chapter 10 — it is foreshadowed here at the pilot's `v0.1.0`, and the enforcement build later bumps that pin to the released `v1.0.0`:

`backend/examples/ch09/apm-policy.warn.yml` — Meridian's pilot policy in `warn`. Target rules use `compilation.target.allow`, not a top-level `targets:`. `apm v0.23.1`

```
name: Meridian Checkout APM Policy
version: "0.1.0"
enforcement: warn                       # measurement phase -- reports, never gates

dependencies:
  allow:                                # only these sources (deny still wins)
    - microsoft/**
    - github/awesome-copilot/**
    - meridian-finance/**
  deny:
    - sketchy-org/**
  require:
    - meridian-finance/meridian-standards#v0.1.0   # resolving this pkg: SKIPPED-needs-network
  require_pinned_constraint: true       # ban bare-branch / wildcard / open-range refs

mcp:
  allow:
    - io.github.github/github-mcp-server           # resolving this MCP: SKIPPED-needs-network
  deny:
    - io.github.sketchy/sketchy-mcp
  transport:
    allow: [http, stdio]                # blocks sse + streamable-http
  self_defined: warn                    # allow | warn | deny

compilation:                            # CORRECTED -- was a top-level `targets:` (a silent no-op)
  target:
    allow: [copilot, claude, cursor]

manifest:
  required_fields: [name, version, description]
  scripts: allow

unmanaged_files:
  action: warn
```

Before trusting a policy, confirm it *parses and registers its rules*. This is the check that would have caught the `targets:` bug — it runs entirely offline against the local file:

`apm policy status` confirms the pilot parses and the corrected target rule registered (`compilation_targets_allowed: 3`; the wrong top-level `targets:` reports `-1`). Runs offline. `apm v0.23.1`

```
$ apm policy status --policy-source ./apm-policy.warn.yml -o json
{
  "outcome": "found",
  "enforcement": "warn",
  "rule_counts": {
    "dependencies_allow": 3,
    "dependencies_deny": 1,
    "compilation_targets_allowed": 3        # corrected key registered 3 targets (was -1)
    # ... other rule_counts omitted
  },
  "extends_chain": []
}                                        # exit 0
```

The pilot policy carries many rules at once, which makes any single exit code hard to read in isolation. To watch the `warn` → `block` dial *cleanly*, take Meridian's highest-risk rule — `require_pinned_constraint` — on its own. The

`meridian-checkout` repo already carries the transitive review skill on a bare `#main` branch from Chapter 7, so that rule has something to catch. The committed `apm-policy.min-warn.yml` is exactly that one rule in `warn`:

`apm audit --ci --policy ./apm-policy.min-warn.yml` — the pin rule *reports* the unbounded ref but the audit still passes. Replays from cache and scans locally, so it runs offline. `apm v0.23.1`

```
$ apm audit --ci --policy ./apm-policy.min-warn.yml
  [>] Replaying install (cache-only)... [+] No drift detected
  ...                                   # baseline + policy checks run
  | [+] | dependency-pinned-constraint | 1 dependency(ies) use unbounded constraints (hint: pin to a semver
range, literal tag, or SHA) (enforcement: warn) |

  [*] All 18 check(s) passed          # exit 0 -- warn MEASURES; it does not fail the build
```

After two sprints of watching that finding (and others) in Code Scanning and remediating the top offenders, the team flips the highest-risk rules to fail closed. In the full pilot that means three tightenings from the `warn` file — the top-level dial to `block`, `mcp.self_defined` to `deny`, and `unmanaged_files.action` to `deny`:

The three enforcement tightenings Meridian made in `backend/examples/ch09/apm-policy.block.yml` when moving from pilot to enforcement — these are the point of the move. (The file also bumps its own `version` and the required-standards pin from the pilot's `0.1.0` to `1.0.0`.) `apm v0.23.1`

```
enforcement: block                    # warn -> block (the whole file now fails closed)

mcp:
  self_defined: deny                  # warn -> deny

unmanaged_files:
  action: deny                        # warn -> deny
```

Now the identical pin violation — run through the same offline audit, using the committed `apm-policy.min-block.yml` to isolate the one rule — fails the build. Same finding, one different exit code — with a bonus: because `block` now fails, it also surfaces the per-dependency detail line that `warn` only tallied. Warn measures the count; block hands you the actionable per-dependency detail right before it gates:

`apm audit --ci --policy ./apm-policy.min-block.yml` — the same finding now *gates*. The only change from the previous run is `enforcement: block`. Runs offline. `apm v0.23.1`

```
$ apm audit --ci --policy ./apm-policy.min-block.yml
  [>] Replaying install (cache-only)... [+] No drift detected
  ...                                   # baseline + policy checks run
  |     | dependency-pinned-constraint | 1 dependency(ies) use unbounded constraints (hint: pin to a semver
range, literal tag, or SHA) |
  |     |                              |   - microsoft/apm-sample-package: bare branch 'main' tracks a
moving tip |

  [x] 1 of 18 check(s) failed         # exit 1 -- block FAILS CLOSED
```

That exit `0` → exit `1` on an unchanged violation is the entire point of the dial. In production the team would *fix* the unbounded ref before flipping (pin the `#main` skill to a tag, exactly as Chapter 6 taught) — the demo holds the violation constant only to isolate the mode change. Two parts of this scenario are real but not exercised offline and are marked `‹ NEEDS NETWORK`: **resolving** the private `meridian-finance/meridian-standards` package and the registry `io.github.github/github-mcp-server` MCP (the *rules* for both parse and enforce; only the specific remote artifacts are not fetched), and the **org-remote discovery** + **tighten-only inheritance merge** that will apply once this file is published to `<org>/.github/apm-policy.yml` and a stricter child `extends:` it.

Recap & next

Recap

- **Security vs. governance are two questions at one gate.** SECURITY asks "is it safe?" (intrinsic, universal); GOVERNANCE asks "is it allowed *here*?" (declared in `apm-policy.yml`, per-org). Both fire pre-disk; both must pass; allowed ≠ safe. Policy governs what *installs*, never what *runs*.
- **Write the verified schema.** Target rules live under `compilation.target.allow` — a top-level `targets:` is **silently ignored** (like any unknown key). `allow: null` = no opinion, `[]` = allow nothing, a list = only these; `deny` wins.
- **Two enforcement points, one rule set.** The install preflight (auto-discovered from the git remote — no `--policy` flag) and the CI gate `apm audit --ci --policy org`. Transitive MCP is re-checked against `mcp.*`; the `mcp.trust_transitive` field parses but does not enforce — the real gate is `--trust-transitive-mcp` (Chapter 8).
- **The dial is `enforcement`.** Same violation: `warn` shows it and exits `0` (measure); `block` fails it and exits `1` (gate). Roll out by measurement — `warn` → read SARIF → remediate → `block` narrowly — never by flipping a switch on day one.
- **Inheritance is tighten-only.** Enterprise → org → repo: allows intersect, denies union, `enforcement` escalates. A repo can only make policy stricter; exceptions are granted upward, at the parent. Local-file `extends:` does not merge — real inheritance needs a remote parent.
- **The engine is early preview at `apm v0.23.1`.** `apm policy` has only `status` and `explain`; pin the CLI before relying on policy as a production gate.

Next

You have now consumed, locked, maintained, secured, and governed agent context — the full four-property arc for a *consumer*. Chapter 10 — Becoming a Producer flips the perspective: packaging and publishing your own reusable skills, prompts, and plugins so other teams install them through the same loop — and shipping them shaped to pass the very GOVERNANCE gate you just wrote.

PART V · PRODUCING & SHARING / CHAPTER 10

Becoming a Producer

Package and publish reusable APM primitives so other teams can consume them through the normal install loop.

Objective

After this chapter you can take agent context that already works in one repo — a prompt, an instruction set, a skill — and **extract it into a standalone, versioned APM package** that other teams install through the same Chapter 5 `apm install` loop, instead of copying the files by hand. You will see why copy-paste reuse quietly recreates Chapter 1's drift, author a producer `apm.yml`, snapshot the package with `apm pack`, and read the crucial **producer/consumer symmetry**: producing adds *no fifth property* — it makes you the *source* of PORTABILITY, REPRODUCIBILITY, PROVENANCE, and GOVERNANCE for everyone downstream.

Concept/Theory

Reuse-by-copy is the Chapter 1 problem wearing a helpful face

Chapter 1 opened on drift: three copies of a review prompt in three home directories, each checking a slightly different threat model. Chapters 4–9 fixed that *inside* one repo — a manifest, a lockfile, a policy gate. But the moment a *second* team wants your good prompt, the old failure is one paste away. Copy the checkout-review prompt into another repo and the two copies begin to diverge immediately: no shared version, no lockfile, no review trail, no way to answer "which prompt produced this behaviour?" The uncomfortable truth this chapter names is that **reuse-by-copy recreates the exact drift the book set out to kill.**

Producing is the fix. It is the act of extracting context that already works — a prompt, a skill, an instruction set — into a standalone, versioned APM package that other repos install through the normal `apm install` loop, rather than copying files by hand. A copy is a point-in-time snapshot that starts drifting the instant it is made; a *dependency* is a pinned, hashed reference that a lockfile can reproduce and `apm update` can advance deliberately (Chapter 7). Producing converts copies into dependencies. APM frames the whole ramp around one promise: "there is no separate 'build pipeline' — the CLI is the build pipeline" (Producer overview).

> ⚠ **Misconception: "we already reuse our prompts — we just copy the files"**
>
> Copying is the anti-pattern, not the solution — it is the Chapter 1 drift problem with a friendly face. Reuse only becomes *governed* reuse when it flows through a versioned package: one name, one version, one reviewable source that every consumer restores byte-for-byte instead of forking.

A package is just a directory with `apm.yml` + `.apm/`

Here is the part that removes the intimidation. There is no packaging tool, no build config, no artifact type you have not already met. "An APM package is a directory with two things: an `apm.yml` manifest and a `.apm/` source tree. Everything else … is generated, optional, or both" (package anatomy). The producer mental model is identical: "a producer package is just a directory with `apm.yml` … `.apm/` … `README.md`" (Producer overview).

You already know this shape from both sides. In Chapter 3 and Chapter 4 you *authored* primitives under `.apm/instructions/` and `.apm/prompts/`; in Chapter 5 you *consumed* other people's packages that had exactly those directories. Producing is not new authoring — it is **relocation into a package shape**. The same source-vs-build split from Chapter 3 is now your responsibility: `.apm/` is what you write and commit; `.github/`, `.claude/`, `.cursor/`,

and `apm.lock.yaml` are generated. “Edit the source under `.apm/` and re-run `apm install` — never edit the deployed copy” (package anatomy).

The symmetry: everything you consumed, you now become the source of

Producing does not add a fifth property to the book's four. It **flips your role**. Every benefit the earlier chapters named as a *consumer* win, the reader now supplies to downstream teams. When Meridian ships `meridian-standards`, its consumers get:

- PORTABILITY — one manifest compiles the same primitives onto Copilot, Claude Code, and Cursor (Chapter 5);
- REPRODUCIBILITY — a lockfile that pins each dependency by `resolved_commit` and fingerprints its source tree by `content_hash`, so a bare `apm install` restores the same bytes; and, *when a package is packed as a bundle*, an extra per-file SHA-256 map on top (Chapter 6);
- PROVENANCE — a pinned, hashed, reviewable source that passes the same install-time scanners from Chapter 8;
- GOVERNANCE — the package installs through the same `apm-policy.yml` gate from Chapter 9; a policy can even *require* it, exactly as Meridian's `require: meridian-finance/meridian-standards` rule did.

This is the spine sentence of the whole book landing: the reader stops being an *npm user* and becomes an *npm publisher* — the same manifest, lockfile, and policy discipline, now pointed outward. APM states the symmetry directly: “every package you publish here installs through the [consumer ramp's] `apm install` command” (Producer overview). The official ramp is a five-rung ladder — *author → compile → preview → pack → publish* — and this chapter walks the two rungs that matter for internal reuse: **author the package**, and **pack it**, then closes the loop by installing the result back into a scratch repo. “You don't need a marketplace to start” (Producer overview).

> • MERIDIAN
>
> A second Meridian service asks for the checkout-review prompt and the fraud-domain engineering rules. The Chapter 1 move would be to paste them into a Slack thread — and start the drift clock. Instead, the platform team makes the pivot from consumer to producer: they lift the checkout-review prompt, the shared engineering instructions, and the secure-payment-review skill out of `meridian-checkout` and into a new package, `meridian-standards`, owned by **Meridian Platform Engineering**. Every one of those files already existed in the checkout repo — producing is relocation, not new work. By the end of the chapter that package will come *back in* to `meridian-checkout` as a pinned dependency, and the version graduates to `1.0.0`.

In APM

A producer package is a directory you already know how to read

Here is `meridian-standards` as a plain directory. Nothing is scaffolded by a special tool: an `apm.yml`, a `README.md`, and one subdirectory per primitive type under `.apm/`. Read it and you will recognise every file from earlier chapters.

The authored `meridian-standards` tree — manifest, README, and three primitives under `.apm/<type>/`. `plugin.json` is *not* here: it is generated by `apm pack`, not hand-written. `apm v0.23.1`

```
meridian-standards/
├── apm.yml                                    # producer manifest (identity + distribution metadata)
├── README.md                                  # what the package is and how to install it
└── .apm/                                      # the SOURCE tree — one directory per primitive type
    ├── instructions/
    │   └── meridian-engineering.instructions.md # shared engineering rules (Money type, idempotency…)
    ├── prompts/
    │   └── checkout-review.prompt.md          # the checkout-review prompt, now reusable
    └── skills/
        └── secure-payment-review/
            └── SKILL.md                        # PCI / payment-review skill
```

Two precisions keep you out of trouble later. First, **author under `.apm/<type>/`**, not root convention directories: `apm pack` is liberal and will also collect a root `skills/` or `instructions/` folder, but `apm install` is stricter and may *not* discover those on the consumer side — so a file that shows up in the bundle can be silently skipped on install, "silently removing those guardrails from consumer environments" (pack a bundle). Second, **commands ship as prompts** — there is no `.apm/commands/` directory; a slash-command is authored as a `.apm/prompts/*.prompt.md` (author primitives).

The producer manifest: identity + type + distribution metadata

A consumer manifest answers "what does *my* repo need?" A producer manifest also answers "*who* is this package, where does it come from, and what is it for?" so other people can evaluate and discover it. It is the *same* `apm.yml` schema you wrote in Chapter 4 — producers simply fill in the optional distribution fields a private consumer manifest can leave out. Here is Meridian's:

`backend/examples/ch10/meridian-standards/apm.yml` — the same schema as a consumer manifest, with the distribution fields filled in. Verified against a fresh `apm init` template + the synthesised `plugin.json`. `apm v0.23.1`

```
name: meridian-standards          # REQUIRED — package identity (its slug when others depend on it)
version: 1.0.0                    # REQUIRED — semver
description: Shared Meridian engineering instructions, prompts, and review skills.
author: Meridian Platform Engineering   # string -> { name: "…" } in the generated plugin.json
license: MIT                      # SPDX id — recorded in the lockfile & SBOM (omitting it -> NOASSERTION)
repository: https://github.com/meridian-finance/meridian-standards
keywords: [meridian, checkout, fintech, ai-agents]
type: hybrid                      # REAL enum, but RESERVED/INERT at v0.23.1 — changes no behaviour today
targets:                          # incl. copilot/claude -> routes `apm pack` to plugin.json mode
  - copilot
  - claude
  - cursor
includes: auto                    # consent to publish everything under .apm/ (the `apm init` default)
```

Only two fields are required to parse — `name` and `version`. Everything else is optional, and unknown top-level keys are *preserved but ignored* (manifest schema §2). The distribution fields are the package's business card: `repository` gives provenance, `license` sets reuse terms, `keywords` feed discovery, and `author` records ownership.

Key	Verdict	What it does for a shared package
`name`	**required**	Package identity — the slug other repos depend on.
`version`	**required** (semver)	The version consumers pin. Meridian ships `1.0.0`.
`description`	optional, read	One-line summary; flows into `plugin.json`; `apm pack` warns if missing.
`author`	optional, read	Attribution. A string becomes `{ "name": … }` in `plugin.json`.
`license`	optional, read	SPDX id. Recorded in the lockfile and SBOM; omitting it records `NOASSERTION`.
`repository`	optional, read	Provenance; mapped into `plugin.json`.
`keywords`	optional, read	Discovery tags; mapped into `plugin.json` (and marketplace tags).
`targets`	optional, **validated**	Harnesses to compile for. Unknown target values are a *parse error*; also routes `apm pack`.
`includes: auto`	optional (the default)	Consent to publish everything under `.apm/`.
`type`	**real enum, but inert**	`instructions` | `skill` | `hybrid` | `prompts`. Reserved for future overrides; no effect at v0.23.1.
(*unknown key*)	silently preserved	No error, no warning — kept in the file, ignored by the CLI.

Which keys `apm` accepts in a producer manifest, and what each one does for a shared package. Verdicts checked against `apm init`, the synthesised `plugin.json`, and the SBOM warnings. `apm v0.23.1`

> **ⓘ Two harmless keys, for opposite reasons**
>
> `type: hybrid` is a **real, documented enum value** that is simply *reserved* at v0.23.1 — package behaviour is inferred from content (the presence of a `SKILL.md`, the `.apm/<type>/` folders), and `type` is not even carried into `plugin.json`. An *unknown* key is the opposite: unrecognised but preserved. The honest one-liner: **`type` is recognised-but-inert; an unknown key is unrecognised-but-preserved — both are no-ops today, for different reasons.** Keep `type` as forward-looking metadata, but do not claim it changes any behaviour now.

The metadata is not decorative: it becomes the package's identity card on the other end. When you do not author a `plugin.json` yourself, `apm pack` synthesises one from `apm.yml` — so `apm.yml` stays the single source of truth (pack a bundle):

The `plugin.json` that `apm pack` generates for `meridian-standards` — mapped straight from the manifest. Note `type` is absent. `apm v0.23.1`

```
{
  "name": "meridian-standards",
  "version": "1.0.0",
  "description": "Shared Meridian engineering instructions, prompts, and review skills.",
  "author": { "name": "Meridian Platform Engineering" },
  "license": "MIT",
  "repository": "https://github.com/meridian-finance/meridian-standards",
  "keywords": ["meridian", "checkout", "fintech", "ai-agents"]
}
```

`apm pack` — snapshot the package into a distributable artifact

`apm pack` is the pack rung of the ladder: it turns your authored primitives into something you can hand off. The one rule to internalise before you run it: **`apm pack` packs the files your last `apm install` deployed — not the raw**

`.apm/` **tree.** So you always `apm install` first. "If `apm pack` reports 'No deployed files found', your `apm.lock.yaml` has no `deployed_files` entries. Run `apm install` first" (pack a bundle).

What `apm pack` *produces* is decided by which blocks your `apm.yml` declares. This routing table is the thing to memorise — it explains why the same command behaves very differently across packages:

If `apm.yml` contains...	`apm pack` writes...	Where
a `dependencies:` block	a **bundle** (plugin.json + primitive dirs + embedded lockfile)	`./build/<pkg>/`, or a `.zip` with `--archive`
a `marketplace:` block	**marketplace** artifact(s)	`.claude-plugin/marketplace.json`
`targets:` including `claude` or `copilot`	**plugin.json** manifest(s), no bundle	in-tree: `.github/plugin/`, `.claude-plugin/`
none of the above	**"Nothing to pack"**	exit `1`

`apm pack` routing, decided by manifest content (confirmed via the "Nothing to pack" error). apm v0.23.1

`meridian-standards` has `targets:` (including `copilot` and `claude`) but **no `dependencies:` block**, so it lands in row 3 — **plugin.json mode**, not a bundle. The safe pre-flight is `apm pack --dry-run --verbose`: it reports exactly what *would* be written and touches nothing on disk. It runs entirely offline:

Install first, then dry-run the pack. For `meridian-standards` that is plugin.json mode: two manifests *would* be written, no bundle. Both commands are offline, exit `0`. apm v0.23.1

```
$ apm install                    # STEP 1 — pack packs what install DEPLOYS, so install first
  [i] Targets: claude, copilot, cursor  (source: apm.yml)
    [+] <project root> (local)
    |-- 1 prompts integrated -> .github/prompts/
    |-- 2 commands integrated -> .claude/commands/, .cursor/commands/
    |-- 3 rule(s) integrated -> 3 targets
    |-- 1 skill(s) integrated -> .agents/skills/, .claude/skills/
  [*] Installed 1 APM dependency in 1.8s.                        # exit 0

$ apm pack --dry-run --verbose    # STEP 2 — pre-flight: what WOULD pack write? (writes nothing)
  Would write plugin manifest to  .github/plugin/plugin.json     # copilot target
  Would write plugin manifest to  .claude-plugin/plugin.json     # claude target
  # cursor is in targets: but gets NO plugin.json — the plugin contract is claude/copilot only
```

⚠Correction: `apm pack --archive -o dist` writes nothing to `dist`

A tempting-but-wrong line is `apm pack --archive -o dist` "to create a shareable archive." For `meridian-standards` it produces **no archive at all**: with no `dependencies:` block, `apm pack` is in plugin.json mode, so `--archive` and `-o` are inert — it writes the two manifests in-tree and leaves `dist/` empty. And the default output directory is **`./build`, not `dist`** in any case. Verified on apm v0.23.1:

```
$ apm pack --archive -o dist --verbose
  [+] Generated plugin manifest:  .github/plugin/plugin.json
  [+] Generated plugin manifest:  .claude-plugin/plugin.json
  # dist/ : EMPTY. No bundle, no .zip. --archive / -o only apply in bundle mode.
```

To get a real `.zip` bundle a package needs a `dependencies:` closure (with *remote* refs — local paths are rejected at pack time). For a primitives-only package like this one, the real distribution channel is a **git ref**, shown in the worked example.

Scaffolds: `apm plugin init` and `apm marketplace init`

You do not have to hand-write the shape. `apm plugin init` scaffolds a fresh producer package — it writes an `apm.yml` plus a `plugin.json`, and adds a `devDependencies:` block (dev-only dependencies that are *excluded* from what `apm pack` ships — the right home for test-only tooling). It is fully offline:

`apm plugin init` scaffolds rung 1 of the ladder — a valid producer skeleton, no network. `apm v0.23.1`

```
$ apm plugin init -y --target copilot     # scaffold a brand-new producer package (offline)
  [+] Created apm.yml                     # standard template + a devDependencies: block
  [+] Created plugin.json                 # exit 0
```

`apm marketplace init` is the discovery side. It adds a `marketplace:` block to `apm.yml`; `apm pack` then builds a `.claude-plugin/marketplace.json` that consumers register with `apm marketplace add <owner>/<repo>`. A **marketplace is a discovery index over git, not a hosted registry** — "a curated index of packages that one repo publishes and many repos install from" (publish to a marketplace). One trap worth knowing: unknown keys *inside* the `marketplace:` block are rejected at parse time — stricter than the top-level manifest, which silently ignores unknowns.

`apm marketplace init` scaffolds a `marketplace:` block (offline). Consumers later reach it with `apm marketplace add .`. `apm v0.23.1`

```
$ apm marketplace init --name meridian-marketplace --owner meridian-finance   # offline, exit 0

# appended to apm.yml:
marketplace:
  owner: { name: meridian-finance, url: https://github.com/meridian-finance }
  build: { tagPattern: "v{version}" }
  outputs: { claude: {} }          # codex optional; each output writes its profile-default path
  packages:
    - name: example-package        # placeholder the scaffold writes; edit to meridian-standards
      description: Human-readable description of the package
      source: meridian-finance/example-package
      version: "^1.0.0"
```

`apm publish` — the registry path (experimental)

APM also has an `apm publish` verb that uploads to a REST registry — but it is **gated behind an experimental feature that is disabled by default**, and publishing itself pushes over the network. For this book it is `SKIPPED-needs-network`. The gate check is verifiable offline, and it tells you exactly what is going on:

`apm publish` is gated behind the disabled `registries` feature. The gate is verified offline; the actual upload is `SKIPPED-needs-network`. `apm v0.23.1`

```
$ apm publish --package meridian-finance/meridian-standards --dry-run
  Error: apm publish requires the experimental registries feature.
  Enable with: apm experimental enable registries.              # disabled by default
```

Even once enabled, `publish` uploads via `PUT /v1/packages/{owner}/{repo}/versions/{version}` — a network operation against a registry API that is still a *v0.2 working draft*. Most teams never need it: a **git ref is the zero-infrastructure distribution channel**, and it is enough for everything this chapter does. The registry route belongs to the enterprise story in Chapter 11.

◆ FOR ENGINEERING LEADERS

This is the step that converts *tribal knowledge* into a **reusable, governed platform asset**. The checkout team's review conventions were previously trapped in one repo and spread by copy-paste; packaged as `meridian-standards`, they become a named, versioned, reviewable dependency that any team installs with one command — and that your Chapter 9 policy can *require*. The payoff compounds the moment a second and third team consume the same approved conventions: a fix or a new rule ships once, as `v1.1.0`, and every consumer picks it up through the normal update loop instead of drifting. The question to put to your platform group: which of our best-practice prompts, instructions, and skills should be *internal packages* rather than wiki pages nobody re-reads?

When to use / pitfalls

How to hand a package off

Producing the package is one thing; getting it to consumers is another. APM distribution is **git-based** today — there is no central public registry — so pick the channel by reach, not habit. For internal reuse, a pinned git ref is almost always the right answer.

Channel	What it is	Reach for when…
git ref (`owner/repo#<ref>`)	The primary APM channel — zero infrastructure, pinned + hashed by the lockfile.	the consumer can reach your git host. *Default choice for internal reuse.*
bundle (`apm pack --archive`)	A `.zip` whose files are pinned by SHA-256; needs a `dependencies:` closure with remote refs.	you must move it by hand — offline, air-gapped, or a customer hand-off.
marketplace	A discovery index over git (`apm marketplace add`), *not* a hosted registry.	many repos need to *discover* many curated packages.
registry (`apm publish`)	Semver-range installs from a REST service — experimental (`registries`).	your org runs a registry proxy (Chapter 11).

Choosing a distribution channel for a produced package.

⚠ **Pitfall: run `apm install` before `apm pack`**

`apm pack` reflects the files your *last install deployed*, read from the lockfile — not the raw `.apm/` tree. A stale or empty lockfile yields "No deployed files found" or "Nothing to pack." The fix is always the same: `apm install` first, then `apm pack --dry-run --verbose` to confirm the file list before you share anything.

⚠ **Pitfall: `-o dist` is not where output lands — the default is `./build`**

The default `apm pack` output directory is `./build`, and `-o` / `--archive` only do anything in **bundle mode** (a `dependencies:` block). Running `apm pack --archive -o dist` on a primitives-only package writes the plugin manifests in-tree and leaves `dist/` empty. Do not assume a `dist/` directory appeared — check the routing table and look in `./build`.

> ⚠ **Pitfall: author under `.apm/<type>/`, not root convention dirs**
>
> `apm pack` will collect a root `skills/` or `instructions/` folder, but `apm install` may not *discover* those on the consumer side — so a primitive can appear in the bundle yet be silently skipped on install, quietly removing a guardrail. The only layout that round-trips through both pack and install is `.apm/<type>/`. Use it for every primitive.

> ⓘ **Recognise, don't misread: `type` and "marketplace"**
>
> `type: hybrid` is real and forward-looking, but it changes *no* behaviour at v0.23.1 — do not teach it as "this makes the package hybrid." And a **marketplace is not a registry**: it is a discovery index layered over git, with no central hosted service behind it. Both are easy to over-read; keep the vocabulary crisp.

Worked example

The producer → consumer round-trip

The real test of a produced package is not that it packs — it is that it *stands alone*. Local primitives can lean on paths, tools, or context that only exist in the origin repo, so the producer's discipline is to install the package into a fresh, throwaway repo and prove it works there. "Install your own package in a scratch repo before declaring it shipped" (Producer overview). This is the consumer ramp from Chapter 5, seen from the producer's chair.

You can do it fully offline with a **local-path source install**. From a fresh scratch repo, install `meridian-standards` by path: APM resolves its `.apm/` source primitives (ignoring the producer's own compiled output), deploys them to the scratch repo's target, and caches the package — exactly as it would for any git-sourced dependency, with no special casing:

The offline round-trip: install `meridian-standards` into a throwaway repo by local path. The extracted context stands alone — exit `0`, no network. `apm v0.23.1`

```
$ apm init -y --target copilot          # a scratch consumer repo, outside meridian-standards
$ apm install ../meridian-standards     # local-path SOURCE install — fully offline
  [*] Validating 1 package...
  [+] …/meridian-standards
  [>] Resolving …/meridian-standards...
    |-- 1 prompts integrated       -> .github/prompts/
    |-- 1 instruction(s) integrated -> .github/instructions/
    |-- 1 skill(s) integrated       -> .agents/skills/
  [*] Installed 1 APM dependency in 1.4s.        # exit 0 — the extracted context stands alone
```

One caveat about that local-path install: APM records the dependency as a machine-specific *absolute path*, which is not committable. For a portable manifest you use a relative `./` path (dev-only) or — the real story — a **git ref**. That is how the loop actually closes: Meridian pushes `meridian-standards` to its private host, tags `v1.0.0`, and `meridian-checkout` depends on it. Because that host is private, the resolve is `SKIPPED-needs-network` in this book — but the manifest change is the whole point:

`meridian-checkout/apm.yml` closing the loop — the extracted conventions return as a pinned dependency, and the version graduates to `1.0.0`. Resolving the private repo is `SKIPPED-needs-network`. NEEDS NETWORK apm v0.23.1

```
# SKIPPED-needs-network: meridian-finance/meridian-standards is a private repo
name: meridian-checkout-agent-context
version: 1.0.0                    # <- the beat: the checkout context graduates from v0.x to 1.0.0
author: Meridian Checkout Team
targets: [copilot, claude, cursor]
dependencies:
  apm:
    - meridian-finance/meridian-standards#v1.0.0   # the extracted conventions, back in as a governed dep
  mcp: []
includes: auto
```

That single pinned line is the payoff of the whole book. The checkout team no longer *copies* its own conventions between services — it *depends* on them. The lockfile pins the exact commit and content hash (Chapter 6); the install-time scanners vet the source (Chapter 8); and the org policy from Chapter 9 can now `require: meridian-finance/meridian-standards`, so *every* Meridian service restores the same approved conventions, byte-for-byte. The four properties the reader consumed for nine chapters are now the four properties they *produce*.

Recap & next

Recap

- **Reuse-by-copy recreates Chapter 1's drift.** Producing turns a working convention into one *versioned, reviewable* package that consumers restore identically — a dependency, not a fork.
- **A package is just a directory** with an `apm.yml` and a `.apm/<type>/` source tree. There is no separate build pipeline; the CLI is the build pipeline. Producing is relocation into a package shape, not new authoring.
- A **producer manifest** is the same schema as a consumer's, with the distribution fields filled in (`description`, `author`, `license`, `repository`, `keywords`). Only `name` + `version` are required; `type` is a real enum but *inert* at v0.23.1; unknown keys are silently preserved.
- `apm pack` **routes by manifest content:** `dependencies:` → a bundle in `./build`; `marketplace:` → a marketplace index; `targets:` (claude/copilot) → `plugin.json` in-tree; nothing → "Nothing to pack." `meridian-standards` is plugin.json mode.
- Mind the corrections: **run** `apm install` **before** `apm pack` (pack packs deployed files); `-o dist` **does not work** — the default output is `./build`, and `-o`/`--archive` only apply in bundle mode.
- Distribution is git-based: a **git ref** is the zero-infra default; bundles are for offline hand-off; a **marketplace is a discovery index, not a registry**; and `apm publish` (experimental `registries`) is `SKIPPED-needs-network`.
- Meridian extracted its checkout-review prompt, engineering instructions, and secure-payment skill into `meridian-standards`, proved it stands alone with an offline round-trip, and closed the loop by depending on `meridian-finance/meridian-standards#v1.0.0` — graduating `meridian-checkout` to `1.0.0`.

Next

Meridian now produces a package its own teams consume — but a package that *many* teams must adopt raises new questions: how do you gate it in CI, serve it in an air-gapped environment, and govern its use across a whole organisation without turning developer setup into a bottleneck? Chapter 11 — Enterprise at Fleet Scale takes the producer and consumer ramps to org scale: `apm audit --ci` gating, a registry proxy, air-gapped installs, and the adoption playbook that makes the Chapter 9 policy stick.

PART VI · AT SCALE & AHEAD / CHAPTER 11

Enterprise at Fleet Scale

Gate and govern APM usage across an organization without turning developer setup into a bottleneck.

Objective

After this chapter you can operate APM across a whole organization — not one repo, but a *fleet*. You will wire `apm audit --ci` into branch protection as a **required, unbypassable** pull-request check (the same integrity gate from Chapter 7 and policy gate from Chapter 9, re-run in CI as defence in depth), stand it up with `microsoft/apm-action`, route dependency traffic through an approved **registry proxy** (or build for a fully air-gapped network), publish and *own* an org `apm-policy.yml` baseline, and roll it all out with a measured adoption playbook. This is the capstone: it introduces no new property. Instead it asks whether the four you already know — PORTABILITY, REPRODUCIBILITY, PROVENANCE / SECURITY, and above all GOVERNANCE — hold for *every* repo, **enforced and operated at scale**. Governance is the star of the chapter: the org's third promise is that "every AI package your developers install is governed by org policy before it touches disk" (Enterprise overview). One honesty note up front, at `apm v0.23.1`: the `apm audit --ci` **baseline is stable**, but three surfaces you will meet here are preview — the `--policy` flag is **experimental**, the dedicated *registry* (distinct from the proxy) is experimental behind a feature gate, and policy enforcement is early preview — so pin the CLI (and `microsoft/apm-action`, latest **v1.10.0**) before you lean on any of them as a production gate.

Concept/Theory

From "can *this* repo install?" to "can *every* repo install?"

A single team gets almost all of APM's value from three files and one habit: `apm.yml`, `apm.lock.yaml`, and `apm install`. Everything in Chapters 5–9 answered a repo-sized question: *does my project install, reproduce, stay current, stay safe, and stay within policy*? An enterprise asks a different, larger question — the shift this whole chapter turns on: not "can *this* repo install?" but **"can *every* repo install safely and predictably, without the platform team in the loop?"**

That reframing surfaces five things the single-team story never forces you to build (Making the case):

- **Repeatable rollout** — a runbook any team follows, not a hero who hand-holds each repo.
- **Policy ownership** — a named, protected owner of the org's `apm-policy.yml`.
- **Audit gates** — a required CI check on every pull request, authoritative regardless of what a developer does locally.
- **Registry strategy** — controlled, mirrored, and (when needed) offline dependency traffic.
- **Exception handling** — a visible, reviewable way to grant a waiver.

The docs' worked figure is a mid-to-large org: *50 repositories, 200 developers, five AI coding tools*. Without central management a predictable failure set emerges — manual config drifting per repo, no audit trail ("what agent configuration was active at release 4.2.1?" has no answer), version drift between developers and CI, onboarding friction, and ungoverned dependencies: "the same problem regulated industries spent a decade solving for application code, now back in a new form" (Making the case). These are the Chapter 1 pains, multiplied by the repo count.

Two boundaries keep the rest of the chapter honest. First, **consuming policy is not owning policy**: a fleet team (including the original pilot) *consumes* the org policy discovered from its git remote — authorship lives in `<org>/.github` behind CODEOWNERS and branch protection, exactly the org-remote model from Chapter 1 and Chapter 9. Second, nothing here is a *new* capability: fleet scale is Chapters 6–10 made **repeatable, owned, gated, and**

measured. The common trap is to think "fleet scale is just the single-repo setup, copied N times" — but N copies without central ownership simply reproduce the drift problem at higher cost. The missing pieces are ownership, gates, a registry strategy, and exceptions, none of which a single repo ever needs.

> ⓘ **There is a threshold — do not over-adopt**
>
> APM is worth this machinery when you *use more than three plugins or MCP servers, your team has more than five developers, you need reproducible CI config, you share standards across repos, or you need an audit trail.* "Below that threshold, manual setup is fine. APM is designed to help when manual management stops scaling" (Making the case). Name the threshold out loud so a small team doesn't stand up an org policy repo it doesn't need yet.

> • MERIDIAN
>
> Meridian's **platform team** spent Chapters 4–10 proving APM on a single pilot, `meridian-checkout`. Now they scale it to **three product groups** — Payments, Onboarding, and Merchant Dashboard. The staff engineer's question is no longer "does checkout install?" but "can all three groups clone, install, and pass CI with the *same* guarantees, without us hand-holding every repo?" Their answer is four coordinated moves, one per fleet need: (1) make `apm audit --ci` a **required** PR check via `microsoft/apm-action`; (2) route all APM dependency traffic through the security org's **approved registry proxy**; (3) publish an org `apm-policy.yml` baseline at `enforcement: block`, owned in `meridian-finance/.github`; and (4) ship an **adoption checklist** so new repos onboard themselves. The worked example runs those four moves.

In APM

> ⓘ **Version note: what is stable vs. preview at v0.23.1**
>
> The `apm audit --ci` **baseline** (eight lockfile checks + the install-replay drift check) and the **registry proxy** (environment-variable driven) are **stable.** Three surfaces are preview: the audit `--policy` flag is tagged `[experimental]` (apm audit reference); the dedicated *registry* (a real Registry HTTP API, distinct from the proxy) is experimental behind `apm experimental enable registries` because the OpenAPM registry API is deferred to v0.2 (Registries guide); and policy *enforcement* is early preview, carried over from Chapter 9. Pin the CLI to `apm v0.23.1` before relying on any preview surface as a production gate.

The fleet gate: `apm audit --ci`

The authoritative enforcer at fleet scale is one command run on every pull request: `apm audit --ci`. It is not a new engine — it is the *same* gate you already know, re-run in CI as **defence in depth.** On a PR it runs the **eight baseline lockfile checks** (`lockfile-exists`, `ref-consistency`, `deployed-files-present`, `no-orphaned-packages`, `skill-subset-consistency`, `config-consistency`, `content-integrity`, `includes-consent`), the **install-replay drift check**, and — *if* an `apm-policy.yml` is discovered from the git remote — the org ● GOVERNANCE checks. Exit code is `0` **clean,** `1` **on any violation** (Enforce in CI).

Why re-run a gate the developer already passed on their machine? Because the install-time gate from Chapter 8 and Chapter 9 protects only the developer's own disk, and it is **locally bypassable**: a developer can pass `--no-policy`, `--force`, or set `APM_POLICY_DISABLE=1`. CI re-runs the identical checks on the pull request itself — and "`--no-policy` does not work here — CI ignores the local bypass flag" (Enforce in CI). Wired into **GitHub Rulesets** as a required status check, a violating PR simply cannot merge (GitHub rulesets). That is what makes an org rule *actually* authoritative on every merge across every repo.

Flag	Job	Fleet note
`--policy <src>`	Explicit policy ref: `org` | `owner/repo` | `https://…` | local path	`[experimental]`. Omit it and APM auto-discovers from the git remote, like `apm install`.
`--no-cache`	Force a fresh policy fetch	**Recommended in CI** — a cached policy file must not mask a same-day org update.
`--no-policy`	Skip policy discovery (baseline + drift only)	Not a bypass of the org gate — CI wires the *unflagged* command; baseline is never bypassable.
`--no-fail-fast`	Run every check even after one fails	Use for full reports and drift sweeps; the default stops at the first failure.
`-f sarif|json`, `-o <path>`	Structured output; write to a file (format inferred from extension)	SARIF feeds Code Scanning. Markdown is **not** supported in `--ci` mode.

The `apm audit --ci` flags that matter in a fleet gate. Verified against apm v0.23.1.

Two properties of the gate are worth holding onto. The **eight baseline checks plus drift always run** and are never bypassable — that is the tie-back to Chapter 6: REPRODUCIBILITY is enforced in CI even when GOVERNANCE is off. And there is **no per-PR override flag, by design**. Exceptions are visible or they do not exist: you either amend `<org>/.github/apm-policy.yml` through normal review (allow-list the package, raise a cap) or lower `enforcement` from `block` to `warn` for that scope — findings still appear in SARIF, they just stop failing the job. "Bypass must be visible in the policy file's history" (Enforce in CI). That is the exception handling the concept promised, made concrete.

`microsoft/apm-action`: the turnkey CI path

`microsoft/apm-action` is the convenience wrapper that installs the CLI, runs `apm install`, and can emit SARIF for Code Scanning (microsoft/apm-action). It is *not* the enforcer — the gate is `apm audit --ci`; the action just stands it up. Pin the action to the major tag `@v1` and pin the CLI with `apm-version:` for reproducible runs. A GitHub Action cannot be executed locally, so every workflow below is `SKIPPED-needs-network` — documented, not run — while the underlying `apm audit --ci` gate it invokes is verified (see the worked example).

`backend/examples/ch11/workflows/apm-audit.yml` — the minimal required-check gate. Made a required status check via GitHub Rulesets, a violating PR cannot merge. NEEDS NETWORK apm v0.23.1

```yaml
# .github/workflows/apm-audit.yml   -- SKIPPED-needs-network (the `apm audit --ci` it runs IS verified)
name: APM audit
on:
  pull_request:
    paths: ['apm.yml', 'apm.lock.yaml', '.apm/**', '.github/**', '.claude/**', '.cursor/**']
jobs:
  audit:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: microsoft/apm-action@v1        # pin to the major tag; installs CLI + runs `apm install`
        with:
          apm-version: '0.23.1'              # pin the CLI for reproducible CI
      - run: apm audit --ci --no-cache       # verified gate: exit 0 clean / 1 on violation
        env:
          GITHUB_APM_PAT: ${{ secrets.APM_PAT }}   # same-org private repos work with zero config
```

For findings to appear inline on the PR diff and in the repo's Security tab, emit SARIF and upload it. The `if: always()` step is load-bearing — SARIF must upload *even when the audit exits* `1`, or the failing run produces no Code Scanning entry:

backend/examples/ch11/workflows/apm-audit-sarif.yml — SARIF for Code Scanning. NEEDS NETWORK apm v0.23.1

```yaml
# .github/workflows/apm-audit-sarif.yml   -- SKIPPED-needs-network (documented)
jobs:
  audit:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      security-events: write                # required to upload SARIF to Code Scanning
    steps:
      - uses: actions/checkout@v4
      - uses: microsoft/apm-action@v1
        with: { apm-version: '0.23.1' }
      - name: Audit
        run: apm audit --ci --no-cache -o apm-audit.sarif   # format inferred from the .sarif extension
        env: { GITHUB_APM_PAT: '${{ secrets.APM_PAT }}' }
      - name: Upload SARIF
        if: always()                          # upload even on exit 1, or a failing run has no alert
        uses: github/codeql-action/upload-sarif@v3
        with: { sarif_file: apm-audit.sarif, category: apm-audit }
```

Two refinements are worth knowing. The default action runs `apm install` first, which **overwrites** managed files and can hide bytes that were hand-edited after the last install; `setup-only: true` puts the CLI on PATH only, so the committed bytes are audited as ground truth (`content-integrity` still verifies each file's SHA-256 against the lockfile). And because `apm audit --ci` is a plain CLI call, the same gate is **vendor-neutral** — it runs in Azure Pipelines, GitLab CI, and Jenkins, not only GitHub Actions (CI/CD integration); `microsoft/apm-action` is just the GitHub-shaped convenience.

ⓘ Beyond CI gating: `gh-aw` runs the same action

The same `microsoft/apm-action` is also how **GitHub Agentic Workflows** (`gh-aw`) consumes APM at *runtime*. A gh-aw workflow adds a `shared/apm.md` import that runs the action to pack the declared packages into a bundle, then restores it before the agent job starts — so the manifest, the `apm.lock.yaml` SHA pins (Chapter 6), and the org `apm-policy.yml` baseline you own here carry straight into an agent's execution environment (gh-aw APM dependencies). Pairing APM's install-time governance with gh-aw's sandboxed runtime — where the model holds no write token — is documented as an enterprise pattern in GitHub's Well-Architected guidance (Governing agentic workflows with gh-aw and APM). Chapter 12 places gh-aw on the wider map as a *consumer* of APM, not a rival.

Registry strategy: proxy and air-gapped

Because "enterprise networks rarely allow agents to reach `github.com` directly," APM routes dependency traffic through composable, environment-variable controls (Registry proxy). The single most important verified fact: **these are environment variables, not `apm config` keys.** At v0.23.1 `apm config` persists only `auto-integrate`, `temp-dir`, and `mcp-registry-url` — so at fleet scale you pin the proxy settings in CI secrets, dev-container env, and shell profiles, "in the same place you pin Python and APM versions," *not* via `apm config set`.

Need	Knob	Note
Allow outbound traffic at the firewall	`HTTPS_PROXY` / `HTTP_PROXY` / `NO_PROXY`	Standard forward-proxy vars — "if `git clone` works through the proxy, `apm install` works too."
Mirror every archive for audit + replay	`PROXY_REGISTRY_URL`	Rewrites every GitHub-hosted download to fetch via the mirror (e.g. an Artifactory GitHub remote).
Authenticate to the mirror	`PROXY_REGISTRY_TOKEN`	Bearer token on proxy requests; independent of `GITHUB_APM_PAT`.
Refuse direct-VCS fallback (mandatory + auditable)	`PROXY_REGISTRY_ONLY=1`	APM refuses to fall back to `github.com`; the lockfile records a `registry_prefix`, and replay aborts on a directly-pinned entry until re-resolved.
Silence the plaintext-token warning on `http://`	`PROXY_REGISTRY_ALLOW_HTTP=1`	Use only inside an isolated network. (Deprecated `ARTIFACTORY_*` aliases still work with a warning.)
Fully air-gapped CI (no egress at all)	`apm pack` on a connected host → restore offline	Chapter 10's bundle, reused as an air-gapped delivery mechanism.

The registry-strategy knobs, with exact environment-variable names. All SKIPPED-needs-network at v0.23.1.

The reproducibility tie-back matters here more than anywhere: **the proxy is not the trust anchor — the lockfile is.** "Every install verifies the `content_hash` recorded in `apm.lock.yaml` regardless of where the bytes came from. A tampered proxy that rewrites archive contents is caught by the lockfile guard, not the cache" (Registry proxy). So routing through Artifactory does *not* weaken `REPRODUCIBILITY`; if anything `PROXY_REGISTRY_ONLY=1` makes the proxy path *mandatory and auditable*. Be honest about coverage, though: the proxy covers `apm install` for GitHub-hosted deps and marketplace fetches, but **not** Azure DevOps deps, **not** MCP servers (a separate registry), and **not** the `apm-policy.yml` fetch (which uses the GitHub API directly). And do not confuse the proxy with the dedicated *registry*: the proxy fronts an upstream git host and is stable; the dedicated registry is a separate, additive package source with no git upstream and is **experimental** (v0.2 API). APM's distribution is **git-based today** — there is no npm-style central registry.

Org policy at fleet scale

The *mechanism* of org policy is already fully covered in Chapter 9 — the schema, the `warn` → `block` dial, tighten-only inheritance. Fleet scale changes only *where it lives* and *who owns it*: the authoritative policy sits in `<org>/.github/apm-policy.yml` behind CODEOWNERS and branch protection, and every consuming repo (the pilot included) only *consumes* it, discovered from its git remote. Past the pilot, the org file is `enforcement: block` with `fetch_failure: block` so a repo whose policy cannot be fetched **fails closed** rather than silently fail-opening. Two parts of this are exactly the parts that need infrastructure a reader's sandbox will not have — **org-remote discovery** and the tighten-only `extends:` **merge** — so both are `SKIPPED-needs-network` here (recall from Chapter 9 that a *local-file* `extends:` does not merge a parent; real inheritance needs an `org` / `owner/repo` / `https://` ref). You author and test the file locally, then land it in `<org>/.github`.

When to use / pitfalls

Roll out in phases, measure by leading indicators

Rolling APM out to a fleet is a **staged program, not a switch**. The official adoption playbook is five phases, each with a single owner, a deliverable, and a gate to clear before advancing (Adoption playbook). The outline's four-word mnemonic — *pilot → measure → standardize → gate* — maps directly onto them:

Phase	Owner	Gate to advance
Discover	Platform team	Shadow `apm install --dry-run` + `apm audit` on representative repos; answer "what breaks if we turn this on tomorrow, and for whom?"
Pilot	One product team + platform	Manifest, lockfile, CI audit, and policy in `warn`; two consecutive weeks of clean pilot CI with every warning triaged.
Harden	Security + platform	Flip `warn` → `block`, add the registry proxy, stand up marketplaces; a fresh repo installs against org policy + proxy with no manual help.
Scale	Product teams (self-service)	Platform is no longer in the critical path; new repos onboard from a checklist.
Sustain	A *named* on-call	Steady state: weekly drift triage, monthly lockfile review, quarterly marketplace refresh.

The five-phase adoption playbook. Each phase buys evidence the next one needs — do not skip phases. From the APM adoption playbook.

Success is measured by **leading indicators, not package counts**. The docs are explicit that "measuring `apm.yml` count and nothing else" is a **vanity metric** — a repo with a manifest but a failing audit or rising drift is not adopted, it is *at risk*. Watch *audit pass rate*, the *drift trend* (findings closed vs. opened), and *marketplace uptake* instead (Adoption playbook).

⚠ Pitfall: `require_pinned_constraint` is direct-only

The `dependency-pinned-constraint` policy check is evaluated against **direct manifest dependencies**, not transitive ones. A *pinned* direct dep (`…#v1.0.0`) that pulls an *unpinned* transitive dep **passes** the check — verified as `All 29 check(s) passed`, exit `0`. Only when the *direct* dep tracks a bare branch does it fail. The install-time `[!] 1 dependency unpinned` warning is therefore **advisory** and does not by itself trip this gate. If your fleet wants transitive pinning too, that is a different lever (lockfile pinning), not this rule.

⚠ Pitfall: the baseline is unbypassable, but `--policy` is experimental

The eight baseline checks + drift always run and are never bypassable — that part is stable and authoritative. The `--policy` flag that adds the org rule checks is `[experimental]` at v0.23.1 (apm audit reference). Rely on the baseline as a hard gate today; treat the explicit-policy layer as version-sensitive and pin the CLI. Also note the check *count* varies with the declared rules and with fail-fast: a passing run enumerates all checks (e.g. `29`), while a failing run stops at the first failure (`1 of 18`). Add `--no-fail-fast` when you want a full report.

⚠ Pitfall: the proxy is env-vars, and never claim a local Action run

The registry-proxy knobs (`PROXY_REGISTRY_*`, `HTTPS_PROXY`) are **environment variables, not `apm config` keys** — pin them in CI secrets and dev containers. Mixed-fleet env skew (some agents with `PROXY_REGISTRY_ONLY`, some without) can cause lockfile drift for three-segment GitLab shorthands, so set them fleet-wide. And because a GitHub Action cannot run on a workstation, **never present a `microsoft/apm-action` result as locally verified** — the action is documented; only the `apm audit --ci` gate it wraps is run in this book.

◆ FOR ENGINEERING LEADERS

Fleet adoption is **change management, not a rollout script**. The pattern that earns trust is *pilot → measure → standardize → gate*: prove it on one product group in `warn`, measure the fleet-wide violations through Code Scanning, standardize the winning setup into an org baseline and a marketplace worth installing from, and only then `block` the highest-risk rules — narrowly. Two failure modes sink programs: flipping `block` on day one (every repo with a pre-existing violation fails at once and you spend the day rolling back), and mandating adoption without offering a marketplace worth installing from — **carrot before stick**. Decide ownership explicitly: "the platform team" is *not* an owner; a **named on-call** is. And measure the right things. The report that persuades a VP is *setup time* (manual 15–60 min per repo → `apm install` in under 30 seconds), *policy warnings trending down*, *risky installs blocked*, *update cadence*, and *developer satisfaction* — **not** the count of repos with a manifest, which is a vanity metric (Making the case; Adoption playbook).

Worked example

Meridian's four moves, in order. Only the first is executable in a reader's sandbox — the CI gate itself — so it is shown running against a local project (offline, one public package, no tokens). The proxy, the workflow, and the org-policy merge each need infrastructure and are marked `SKIPPED-needs-network`.

Move 1 — the required gate (RUNNABLE offline)

A clean project — one *pinned* public dependency plus its committed lockfile — passes the gate. The clean run is the eight baseline checks plus the drift check: **nine checks**, exit `0`. (The `Could not determine org…` line is benign: a scratch directory has no git remote, so org auto-discovery is skipped.)

`apm audit --ci` on the clean project — nine checks pass, exit 0. This is the gate the required PR check runs. Replays from cache and scans locally, so it runs offline. Transcript abbreviated for space. `apm v0.23.1`

```
$ apm audit --ci
  [!] Could not determine org from git remote; enforcement skipped (set policy.fetch_failure_default=block
in apm.yml to fail closed)
  [>] Replaying install (cache-only)...  [+] Replayed 2 package(s)  [+] No drift detected

                         [>] APM Policy Compliance
  | [+] | lockfile-exists          | Lockfile present                                |
  | [+] | ref-consistency          | All dependency refs match lockfile              |
  | [+] | deployed-files-present   | All deployed files present on disk              |
  | [+] | no-orphaned-packages     | No orphaned packages in lockfile                |
  | [+] | skill-subset-consistency | Skill subset selections match lockfile          |
  | [+] | config-consistency       | No MCP configs to check                         |
  | [+] | content-integrity        | No critical hidden Unicode or hash drift detected |
  | [+] | includes-consent         | No local content deployed -- includes … skipped  |
  | [+] | drift                    | no drift detected against lockfile              |

  [*] All 9 check(s) passed          # exit 0 -- 8 baseline checks + drift
```

Now the case that makes the required check *authoritative*: a `block` policy (`enforcement: block`, `require_pinned_constraint: true`) against an **unpinned direct dependency** on a bare `#main` branch. The `dependency-pinned-constraint` check fails and the gate exits `1` — a required PR check would block the merge, and no local `--no-policy` can rescue it in CI:

`apm audit --ci --policy ./pol-block.yml` when a direct dependency tracks a bare branch — the check fails, exit 1. Runs offline against a local project + policy file. Transcript abbreviated for space. apm v0.23.1

```
$ apm audit --ci --policy ./pol-block.yml   # enforcement: block, require_pinned_constraint: true
  [>] Replaying install (cache-only)... [+] No drift detected
  ...                                        # baseline + policy checks run
  |     | dependency-pinned-constraint | 1 dependency(ies) use unbounded constraints
  |     |                              | (hint: pin to a semver range, literal tag, or SHA) |
  |     |                              |   - microsoft/apm-sample-package: bare branch 'main' tracks a
moving tip |

  [x] 1 of 18 check(s) failed          # exit 1 -- the required check blocks the PR
```

The same block policy against a *pinned* direct dep — even one that pulls an unpinned *transitive* — passes, because `require_pinned_constraint` is direct-only. Note the higher check count: with no failure, fail-fast never trips, so every check is enumerated:

Same policy, but the direct dep is pinned (`#v1.0.0`) — the unpinned transitive does not trip the direct-only rule; all checks pass, exit 0. Runs offline. apm v0.23.1

```
$ apm audit --ci --policy ./pol-block.yml   # DIRECT dep pinned to #v1.0.0; a transitive dep is unpinned
  | [+] | dependency-pinned-constraint | All dependencies use pinned constraints |
  ...
  [*] All 29 check(s) passed          # exit 0 -- direct-only rule; count higher because nothing fails
                                      #          (fail-fast never trips; use --no-fail-fast for a full
report)
```

Move 2 — make it required in CI (SKIPPED-needs-network)

Meridian stands the verified gate up on every PR in all three product groups with `microsoft/apm-action`, then makes the job a **required status check** via GitHub Rulesets so a violating PR cannot merge. The workflow is the one shown earlier (`backend/examples/ch11/workflows/apm-audit.yml`); the SARIF variant surfaces each finding on the PR diff. Both are documented, not run — but the `apm audit --ci` they invoke is the command proven in Move 1.

Move 3 — route traffic through the approved proxy (SKIPPED-needs-network)

Meridian's security org already mandates Artifactory for npm and PyPI, so APM joins the same operating model. The platform team pins these environment variables in CI secrets and the shared dev container — *not* in `apm config` — so the whole fleet resolves identically:

Routing APM dependency traffic through the approved registry proxy. These are environment variables, not `apm config` keys. Needs an Artifactory / private host to run. ‹ NEEDS NETWORK apm v0.23.1

```
# SKIPPED-needs-network: pin in CI secrets / dev-container env, not `apm config`.
export HTTPS_PROXY="http://proxy.meridian.example:8080"
export PROXY_REGISTRY_URL="https://artifactory.meridian.example/artifactory/github-remote"
export PROXY_REGISTRY_TOKEN="$ARTIFACTORY_TOKEN"   # bearer token; independent of GITHUB_APM_PAT
export PROXY_REGISTRY_ONLY=1                        # refuse direct github.com fallback (mandatory +
auditable)

apm install                                        # every archive fetched via the mirror;
                                                   # content_hash from apm.lock.yaml still verified (Ch6)
```

A fully air-gapped group would instead receive a pre-built bundle: `apm pack` on a connected host (Chapter 10), restored offline. Either way, integrity is anchored to the lockfile's `content_hash`, so the mirror is a routing and audit convenience, never a new trust anchor.

Move 4 — publish and own the org baseline (SKIPPED-needs-network)

Finally the platform team lands the org `apm-policy.yml` in `meridian-finance/.github`, behind branch protection. This is the Chapter 9 schema, relocated and set to fail closed; the three product-group repos *consume* it from their git remote. Authoring and testing happen locally; the org-remote discovery and tighten-only `extends:` merge that make it fleet-wide need infrastructure, so they are `SKIPPED-needs-network`:

`meridian-finance/.github/apm-policy.yml` — the fleet baseline, owned behind CODEOWNERS + branch protection; consuming repos only consume it. Org-remote discovery + `extends:` merge need infrastructure. NEEDS NETWORK apm v0.23.1

```
# meridian-finance/.github/apm-policy.yml   -- SKIPPED-needs-network (org-remote discovery + extends:
merge)
name: meridian-org-baseline
version: "1.0.0"
enforcement: block              # past the pilot: fail closed on violations
fetch_failure: block            # if the policy can't be fetched, fail closed (do not fail-open)
dependencies:
  require_pinned_constraint: true   # fires on a bare-branch DIRECT dep (Move 1) -- direct-only
  allow:                            # only these sources (deny still wins)
    - meridian-finance/**
    - microsoft/**
    - github/awesome-copilot/**
  deny:
    - sketchy-org/**
compilation:
  target:
    allow: [copilot, claude, cursor]   # target rules live HERE, not a top-level `targets:` (Ch9)
```

With the baseline in `warn` first, Meridian watches Code Scanning across the three groups for two sprints, remediates the top offenders, then flips to `block` — the same measured rollout as Chapter 9, now fleet-wide, with the pilot repo as the canary. Their status report to leadership is *audit pass rate* and *drift trend*, not "repos with a manifest."

Recap & next

Recap

- **The question changes at scale.** From "can *this* repo install?" to "can *every* repo install safely and predictably?" A fleet needs five things one team never forces: repeatable rollout, policy *ownership*, audit *gates*, a registry strategy, and exception handling. This is the capstone for GOVERNANCE — enforced and operated across the org.
- **The fleet gate is `apm audit --ci`.** The same integrity gate (Chapter 7) and policy gate (Chapter 9), re-run in CI as defence in depth: eight baseline checks + drift (+ discovered policy), exit `0`/`1`. Made required and unbypassable via GitHub Rulesets — local `--no-policy` does not work in CI. Stand it up with `microsoft/apm-action@v1`; it is vendor-neutral.
- **Registry strategy is environment-variable driven.** `HTTPS_PROXY` + `PROXY_REGISTRY_URL` + `PROXY_REGISTRY_ONLY=1` route and mirror traffic; `apm pack` serves air-gapped networks. Integrity stays anchored to the lockfile `content_hash` (REPRODUCIBILITY), not the proxy — the proxy is never the trust anchor.
- **Adoption is change management.** Discover → Pilot → Harden → Scale → Sustain, each with an owner, a deliverable, and a gate. Measure by *leading indicators* — audit pass rate, drift trend, marketplace uptake, setup time — not by manifest count, which is a vanity metric. Carrot before stick; a named on-call, never "the platform team."
- **Mind the preview edges at `apm v0.23.1`.** `require_pinned_constraint` is direct-only; the baseline is stable and unbypassable but `--policy` is experimental; the proxy knobs are env-vars, not `apm config`; and never claim a locally-run Action result.

Next

You have now consumed, locked, maintained, secured, governed, produced, and *operated* agent context across a fleet — the full arc from one repo to an organization. Chapter 12 — The Landscape & What's Next steps back to place APM among the standards it builds on (AGENTS.md, Agent Skills, MCP, OpenAPM v0.1) and the roadmap ahead — including the dedicated registry API this chapter flagged as experimental — so you can decide what to adopt, watch, or build around.

PART VI · AT SCALE & AHEAD / CHAPTER 12

The Landscape & What's Next

Place APM in the market and standards landscape, then decide what to adopt, watch, or build around.

Objective

After this chapter you can place APM on a *grounded map* of the still-forming category around it — using the book's four properties (PORTABILITY, REPRODUCIBILITY, PROVENANCE / SECURITY, and GOVERNANCE) as the map's coordinates rather than marketing claims. You will name the shared **standards layer** APM builds on (SKILL.md, AGENTS.md, MCP) and the broader **OpenAPM** bid it advances; compare APM honestly against `gh skill`, `orthogonalhq/apm`, `TheLarkInn/aipm`, and `vercel-labs/skills`; and make the **adopt / watch / build-around** call as a decision about *scope and governance*, not install ergonomics. One discipline frames the whole chapter: this is a snapshot dated **2026-07-01**, and every tool named below — APM included — is **pre-1.0 and moving**, so treat the map as a map, not a verdict.

Concept/Theory

A young category that has not settled

"Agent context management" is a genuinely new tool category — barely a year old as a named idea. APM was introduced on the GitHub Blog in October 2025 (How to build reliable AI workflows with agentic primitives and context engineering) and only reached the `microsoft/apm` organization in early 2026 (microsoft/apm). A category this young has **no settled winner and no frozen interfaces**: standards, registries, and even the *scope* of a package (skills only, or every primitive type?) are still being negotiated in public.

So the responsible closing move for this book is not a prediction of who wins. It is a **grounded map**: what exists, what each tool optimizes, and where the seams are. The four properties you have carried since Chapter 1 are the map's coordinate system. The honest way to compare two agent-context tools is not "which `install` command is shorter," but *which of the four properties each one actually delivers, and across how many primitive types*. That reframing is this chapter's whole job.

Keep the distinction between a **map** and a **prediction** in view. A market report ranks vendors and forecasts share; a grounded map plots tools against stable axes so you can locate *your own* need on it. The goal is that you leave able to place a new tool you meet next year — not to memorize today's leaderboard.

> ⚠ **Misconception: "there's a standard package manager for agents now, and APM is it"**
>
> Not yet. The category is contested. A broad specification (OpenAPM) and a widely adopted narrow format (SKILL.md) are both in play, and several capable installers exist alongside APM. APM is *a* strong, broad answer — not *the* settled answer. Overclaiming here would cost you exactly the credibility the rest of this chapter depends on.

The standards layer underneath the tools

Beneath the competing command-line tools sits a shared **standards layer** that most of them build on. APM's own framing is that it is "built on open standards" (microsoft/apm) — it does not compete *with* these formats, it **consumes and composes** them:

- **SKILL.md / Agent Skills** (agentskills.io) — the community format for packaging a reusable skill, adopted across a large number of agents. APM implements it for its `skill` primitive.

- **AGENTS.md** (agents.md) — the shared convention for the agent instruction file, now maintained as an open community standard (its stewardship moved under the Linux Foundation's Agentic AI Foundation in late 2025; see agents.md). APM builds on it as a foundation.
- **MCP** — the Model Context Protocol for tool and data servers (modelcontextprotocol.io). APM declares and governs MCP servers as a primitive rather than reinventing them.
- **OpenAPM v0.1** (the OpenAPM spec) — Microsoft's own *broader* bid. Where SKILL.md standardizes one primitive, OpenAPM tries to standardize the whole **manifest + lockfile + policy** triad (`apm.yml`, `apm.lock.yaml`, `apm-policy.yml`), dependency resolution, and the primitive type system. It is an editor's Working Draft with no backward-compat guarantee before v1.0, and — the fact that matters most this chapter — it defines **no central registry today; the registry API is on the public roadmap (OpenAPM v0.2)**.

Hold two distinctions apart, because the rest of the chapter leans on them. First, **standard vs. tool**: a *spec* (SKILL.md, OpenAPM) defines a format; a tool (APM, `gh skill`) implements one. Second, **narrow vs. broad**: SKILL.md standardizes a single primitive (the skill), while OpenAPM standardizes the packaging system across every primitive. That makes SKILL.md and OpenAPM not rivals for one slot but **two live bids at different scopes** — a broad spec that can *consume* the narrow one. The open question is *convergence*: whether the broad spec absorbs or aligns with the widely adopted narrow one over time. That is something to watch, not to bet on.

The standards layer is also *why* the properties are possible at all. Shared formats are what make PORTABILITY real: a skill written to SKILL.md or an instruction file written to AGENTS.md moves across harnesses because the *format* is shared, not the tool. OpenAPM extends that same shared-format guarantee to REPRODUCIBILITY (a normative lockfile) and GOVERNANCE (a normative policy file). Standards are the substrate; the four properties are what they buy.

In APM

Where APM sits on the map

APM's distinctive position has two halves, and honesty requires stating both. The first half is its strength: among the tools in this space, APM has the **broadest primitive coverage** — the seven types you met in Chapter 3 (instructions, prompts, skills, agents, plugins, MCP servers, and hooks) — *and* it ships enterprise **governance** in `apm-policy.yml` (Chapter 9), which no skills-only installer offers. In the language of the four properties, APM is the tool that carries *all four* across *all* primitive types, not one property over a single slice.

The second half is the gap: APM has **no central, searchable registry today.** Distribution is git-based — "any git repository is a valid APM package" (What is APM?) — while several narrower rivals already ship a registry or GitHub-backed discovery. The registry HTTP API (publisher identity, search, yank/deprecate) is on the public roadmap (**OpenAPM v0.2**), and its progress is tracked on microsoft/apm. This is the category's biggest open question for APM, and the book has said so plainly since Chapter 2 — it is not a footnote.

Why the missing registry is not a missing core

Here Chapter 2 pays off. The registry is the **most optional** part of a package manager; the load-bearing parts are the *manifest* (declared intent) and the *lockfile* (a pinned, verified result). npm, pip, and Cargo all resolved git and path sources long before a central index mattered. So git-based APM *already* delivers all four properties today: a team gets PORTABILITY from the manifest, REPRODUCIBILITY from the lockfile (Chapter 6), PROVENANCE / SECURITY from content-hash pinning and install-time scanning (Chapter 8), and GOVERNANCE from policy (Chapter 9) — none of which needs an index.

The clean line to draw: **"no registry" limits *discovery*, not *management*.** What git-based APM does not yet give you is easy search and browse of third-party packages. That is a real limitation — but it is about the convenience of *finding* packages, not the ability to declare, pin, verify, and govern them. Conflating the two is the trap; it is the Chapter 2 misconception ("a package manager is basically a downloader with a registry") in new clothes.

◆ FOR ENGINEERING LEADERS

The make-vs-adopt decision is a **scope-and-governance** decision, not an install-UX comparison. A narrow skill installer can be exactly right for a single developer who only needs skills and wants a registry today. A regulated, multi-team organization that has already committed to the four properties needs the manifest + lockfile + **policy** + audit story across *every* primitive type — which today points to APM, eyes open about the git-based distribution and the registry that has not landed yet. Building that in-house means re-deriving OpenAPM; adopting APM means inheriting it. The strategic variable to track is **OpenAPM v0.2** (registry, publisher identity, yank/deprecate), because it decides whether the category consolidates on a broad standard or fragments across narrower tools. Adopt for the contract you can rely on now; budget *attention*, not a blocking dependency, for the registry.

When to use / pitfalls

The tools, side by side

Several tools overlap with APM, and none of them is a strawman. Each is real and actively developed; each optimizes a *subset* of the four properties over a *narrower* primitive set. Read the table below as "which properties, over how many primitive types" — not "which install command is shortest."

⚠ **Pre-1.0 caveat: this is a snapshot, not a verdict**

Every tool in the next table is pre-1.0 and moving. This is a snapshot dated **2026-07-01** (APM `apm-cli` v0.23.1; OpenAPM v0.1 Working Draft). APM itself ships roughly one to three releases a week (microsoft/apm), with no backward-compat guarantee until v1.0 (the OpenAPM spec), and its rivals are just as early. Re-verify each cell against its primary source before you rely on it.

Tool (2026-07-01)	Owner · language	Manifest	Lockfile	Central registry	Primitive coverage	Governance / policy
Microsoft APM (*this book*) `microsoft/apm`, `apm-cli` v0.23.1	Microsoft · Python	`apm.yml`	`apm.lock.yaml` (versions + content hashes)	**None yet** — git-based; registry API on the public roadmap (OpenAPM v0.2)	**Broadest** — all seven: skills, prompts, instructions, agents, plugins, MCP servers, hooks	`apm-policy.yml`, install-time, tighten-only inheritance
`gh skill` (*official GitHub*) GitHub CLI v2.90.0	GitHub · Go	none (provenance in SKILL.md)	none	GitHub-as-registry	Skills only	none (relies on GitHub supply chain)
`orthogonalhq/apm` (*same CLI name*) github.com/orthogonalhq/apm	Orthogonal (indep.) · Rust + Next.js	none	`apm-lock.json`	**Public, searchable** (apm.orthg.nl)	Skills only	none
`TheLarkInn/aipm` github.com/TheLarkInn/aipm	Sean Larkin (independent) · Rust	`aipm.toml`	content-addressable store (partial)	planned	skills, agents, MCP servers, hooks (Claude-focused)	none stated
`vercel-labs/skills` (*"npx skills"*) github.com/vercel-labs/skills	Vercel Labs · TypeScript	none	none	git URLs + `skills.sh` index	Skills only (broad agent support, ~72 per its repo)	none

Agent-context tools by the four-property axes — manifest, lockfile, registry, primitive coverage, and governance. Versions as of 2026-07-01; every tool is pre-1.0.

Mapped onto the four properties, the table reads cleanly. Every tool delivers some PORTABILITY — that is the price of entry — but only APM and `aipm` carry it across every primitive type. REPRODUCIBILITY needs a lockfile, which APM, `orthogonalhq/apm`, and (in plan) `aipm` have but the two skills-only installers do not. On PROVENANCE / SECURITY, `gh skill` has the cleanest supply-chain story (immutable releases, tree-SHA) via GitHub-as-registry (the gh skill changelog), while APM leans on content-hash pinning plus install-time scanning. And on GOVERNANCE, APM is effectively alone: no skills-only installer offers an org-level allow/deny gate.

When a narrow installer is enough — and when it isn't

They do not all do the same thing, so "pick the one with the nicest CLI" hides the difference that actually matters. Use the scope of your need to decide:

- **A narrow skill installer is plenty** when an *individual* workflow only needs skills and values a registry or clean provenance *today* — `gh skill` for GitHub-native provenance, `orthogonalhq/apm` for a searchable index, `vercel-labs/skills` for the widest agent reach. A skills-only tool cannot express a project's prompts, instructions, agents, and MCP servers as one governed, locked dependency graph, but if you never needed that graph, the ceremony would be overkill.
- **You need the full manifest / lockfile / policy / audit story** when context is shared, plural, and answerable to someone — a regulated or multi-team context where "which agent configuration was active at release 4.2.1?"

must have an answer, and where an organization decides which sources and MCP servers are allowed. That is APM's coverage, and it is the shape Meridian has been building toward for eleven chapters.

- **"Build around"** — a thin wrapper or an internal registry proxy (Chapter 11) — is the middle path for a team that wants APM's contract *and* internal discovery before the official registry lands.

> **ⓘ The name collision, revisited: it is not hypothetical**
>
> Chapter 2 warned that several unrelated tools share the name "apm." This chapter is where that warning grows teeth. `orthogonalhq/apm` is not a thought experiment — it is a real, actively developed Rust CLI with the *same command name* and, pointedly, the **public registry APM itself lacks** (apm.orthg.nl). A teammate who searches "apm install" can very plausibly land on it, or on `TheLarkInn/aipm`, and install a different tool with a different manifest, scope, and trust model. The fix is the same three-part tell from Chapter 2: this book's APM is `apm-cli` from `microsoft/apm` with the `apm.yml` / `apm.lock.yaml` / `apm-policy.yml` triad. Worth noting without defensiveness: `aipm`'s own design notes ship a point-by-point critique of Microsoft APM (TheLarkInn/aipm) — a useful, non-hyped signal of where a serious competitor thinks APM is over-broad.

Two more tools are adjacent rather than head-to-head, and worth knowing so you do not miscategorize them: `anthropics/skills` is the SKILL.md reference implementation and a content library, *not* a manager; and the MCP Registry and Smithery are MCP-server indexes APM can *consume* rather than replace.

A different relationship: tools that *consume* APM

Not every neighbour on the map is a rival. The most consequential one is a *consumer*: GitHub Agentic Workflows (`gh-aw`), a GitHub Next / Microsoft Research framework that compiles a markdown workflow file into a deterministic GitHub Actions run. The rivals above answer "should I use APM or something else to *manage* context?" `gh-aw` answers the question that comes *after* that one: once context is managed, **who runs it?** It does not reimplement packaging — it **imports APM**. A workflow adds a `shared/apm.md` import that runs `microsoft/apm-action` (the turnkey action from Chapter 11) to pack the declared packages into a bundle, then restores that bundle before the agent starts (gh-aw APM dependencies; the APM · gh-aw integration guide).

Through the four properties, the division of labour is clean — and it is why a consumer belongs on this map at all. APM supplies `PORTABILITY`, `REPRODUCIBILITY` (the same `apm.lock.yaml` SHA pins from Chapter 6, whose diffs are reviewable in the pull request), install-time `PROVENANCE / SECURITY` (Chapter 8), and org `GOVERNANCE` via `apm-policy.yml` (Chapter 9). `gh-aw` contributes the one thing APM deliberately does *not* own: a **sandboxed runtime** that separates the model — which reads untrusted input and holds no write token — from the deterministic step that is allowed to write, so a prompt-injection payload cannot become a pushed commit. GitHub documents the pairing as an enterprise pattern in its Well-Architected guidance (Governing agentic workflows with gh-aw and APM), including the integration's `isolated` import mode, which hides repo-level config from a consumed skill so its *publisher* — not a compromised repo — controls what the agent is told to do. This is the honest shape of the category: APM is not the whole stack, it is the *context layer* other runtimes build on.

Worked example

• MERIDIAN

Meridian's last beat is not a new `apm.yml` — it is a *decision*. Preparing the platform group's quarterly review, you write a short "what we standardize, what we watch" note. After the journey from Chapter 1's drift — three copies of `review.prompt.md`, mismatched Copilot and Cursor rules, Anika onboarding by wiki-page-and-Slack — to the fleet rollout in Chapter 11, the team already relies on all four properties. The note simply makes the posture explicit:

- **Adopt now.** APM for `apm.yml` manifests, `apm.lock.yaml` lockfiles, the org `meridian-finance/.github/apm-policy.yml`, and the internal `meridian-standards` package (Chapter 10) — the contract the team already relies on across Copilot, Claude Code, and Cursor.
- **Watch, don't block on.** The OpenAPM **v0.2 registry** story; whether SKILL.md and OpenAPM *converge*; and whether an internal registry proxy (Chapter 11) should bridge the discovery gap before v0.2 ships. None of these is a reason to delay adoption.
- **Guard against.** The name collision. Internal setup docs and search links say "**Microsoft APM**" / `microsoft/apm`, never a bare "apm," so nobody installs `orthogonalhq/apm` or `aipm` by mistake.

The note ends not with a victory lap but with the questions the team will carry into next quarter: *if the official registry slips, do we stand up our own proxy or wait? If SKILL.md and OpenAPM diverge instead of converging, which do our internal packages track?* Those are open on purpose — the same honest posture the whole book has kept.

Recap & next

Recap — the whole arc

This book made one wager: *agent context is where application code was before npm, pip, and Cargo* — scattered, drifting, "works on my machine." Every chapter added one layer, and each layer protected one of the four properties:

- PORTABILITY — the `apm.yml` manifest and harness targets (Chapters 4–5) turned Meridian's four scattered per-harness files into one declared source of truth.
- REPRODUCIBILITY — the `apm.lock.yaml` lockfile (Chapter 6), forced by a CI break, made restores byte-for-byte; Chapter 7 made change deliberate.
- PROVENANCE / SECURITY — install-time scanning and content-hash pinning (Chapter 8) blocked a sketchy transitive MCP server before it touched disk.
- GOVERNANCE — `apm-policy.yml` (Chapter 9) and its warn → block rollout, then fleet-scale ownership and CI gating (Chapter 11), turned rules-in-heads into an enforced, owned contract — after Meridian crossed the ramp to *publishing* `meridian-standards` (Chapter 10).

And this chapter turned those four properties into a **yardstick**: a grounded map of a young category, where APM sits as the broad-coverage, policy-carrying, git-based option among narrower installers — the make-vs-adopt call reframed as scope and governance, not install UX, and the Chapter 2 name collision confirmed as real, not hypothetical.

Open questions

This is the last chapter, so there is no next one — the "next" is **yours.** The category is still settling, and the honest close is a set of open questions to carry into your own adoption, not a sales pitch:

- **Does the registry land, and when?** OpenAPM v0.2 — the registry HTTP API, publisher identity, yank/deprecate — is the roadmap milestone that decides whether discovery stops being APM's rough edge (microsoft/apm).
- **Do the standards converge or fragment?** Whether the broad OpenAPM bid aligns with the widely adopted SKILL.md format — or the two drift apart — will shape which packages stay portable across tools.
- **Does the category consolidate?** Whether one broad standard emerges or the space splinters across narrower installers is genuinely undecided at v0.23.1 — and everything above is a 2026-07-01 snapshot that will move.

Your move is the one Meridian made: **adopt APM for the contract you can rely on now** — the manifest, the lockfile, policy, and your own internal packages, delivering all four properties over git today — while you **watch** the registry and the standards, and **guard against** installing the wrong "apm." Start where the drift hurts most (`microsoft/apm`, the docs), declare one manifest, and carry the open questions above into your own adoption.