12 — Extensibility

🕒 state of the art 2026-07 · revised 2026-07-26 · 📖 ~11 min read

Learning objectives

By the end of this chapter, you should be able to:

1. **Explain** why extensibility is "open for extension, closed for modification" — extension points instead of forking;
2. **Distinguish** the four extension axes (hooks · commands/skills · plugins · providers) and what each one solves;
3. **Compare** the three ecosystem strategies — depth, packaging, interoperability;
4. **Evaluate** extension code as an attack surface (the *trust triangle*) and the defenses (scanning, trust envelope, managed settings, least-privilege);
5. **Implement** a pre/post-tool hook subsystem with the hook's return value as the control channel in harness-zero (step 11).

The problem

No harness covers every workflow; extensibility decides whether the user **adapts** the harness or **abandons** it. The established axes:

1. **Hooks** — user code intercepting the lifecycle (before/after a tool, compaction, session).
2. **Skills / custom commands** — capabilities packaged as markdown/config, loaded on demand.
3. **Plugins / extensions** — distributable packages aggregating tools, commands, hooks, and config.
4. **Model providers** — the most strategic extension: does the harness work with any model, or is it one model's showcase?

The rule uniting all four is old: **open for extension, closed for modification** — the user extends without editing (or forking) the core.

Scientific foundations

Honest editorial record (Principle I): **there is no academic canon of "agent harness extensibility"** — it is a real gap. The durable citations come from the classic software engineering of extensible architectures and from plugin-ecosystem security, which transfer directly.

- **Extension points, not forks** — the open-closed principle (Meyer, 1988; Martin, 1996) and Eclipse's plug-in architecture (Birsan, *ACM Queue* 2005) provide the foundation — and the "*plug-in hell*" warning: poorly designed extension points become debt. Decision: expose explicit *seams* (events, well-known directories), not ad-hoc points.

- **Minimal core, pluggable extensions** — the Microkernel pattern (Buschmann et al., *POSA* v.1, 1996) and its agentic incarnation, AIOS, arXiv 2403.16971 (a kernel that isolates scheduling/memory/tools from agent applications), support the "harness as microkernel" posture: a small core that serves as a socket.
- **Mechanism × policy** — Hydra (Levin et al., SOSP '75) is the origin of "separate mechanism from policy". Translated: the harness provides the *mechanism* (invoking a tool, dispatching a hook, loading a provider); the *extension* provides the policy. That is why adding a model provider can be "writing a file".
- **Third-party extensions are not trustworthy** — the best on-topic citation is LLM (Large Language Model) Platform Security: ChatGPT Plugins, arXiv 2309.10254 (AIES '24): a platform/plugin/user *trust triangle* with concrete exploits (session hijacking via a malicious plugin). And the empirical base for over-privilege comes from browser-extension security (Barth et al., NDSS '10: 88% of extensions request more power than they need). Decision: least-privilege + isolation + verification — the same argument as ch. 06's *tool poisoning*.

(Full bibliography and pointers: `livro/bibliografia.md`.)

Industry sources

- **Hooks: exit code as the control channel** — Claude Code's hooks expose ~31 lifecycle events (`PreToolUse`, `PostToolUse`, `Stop`, `SessionStart`, `UserPromptSubmit`, `PreCompact`, `SubagentStop`…) where the harness runs user commands; the **exit code is the channel** (0 = proceed / JSON on stdout with allow-deny-ask; 2 = block with stderr fed back to the model). Decision: teams enforce policy (block `rm`, redact `.env`, auto-lint) **deterministically and without patching the harness**. And Codex implements the same pattern independently (hooks + `allow_managed_hooks_only` for enterprises) — hooks are a **cross-vendor** standard, not one vendor's quirk.
- **Plugin = the packaging unit; marketplace = the catalog** — the Claude Code plugin model: a plugin aggregates skills, subagents, hooks, MCP (Model Context Protocol), and LSP (Language Server Protocol) into an installable package (`/plugin install name@marketplace`); a marketplace is a git repo with `.claude-plugin/marketplace.json`. Installs at user/project/local/**managed** scope, with **pinning to SHAs** and a two-tier trust model (curated official marketplace + community with security triage). Decision: third-party extension becomes distributable **and governable** without forking.
- **Custom commands became a file-drop (and AGENTS.md is the open standard)** — in Claude Code, slash commands were absorbed by skills: dropping a file into `.claude/commands/` or `.claude/skills/` creates the command, with no registration or build. And AGENTS.md became the **open, multi-tool** config format — read by Codex, Cursor, Cline, Windsurf, Gemini CLI, and Claude Code. Decision: the extension point is "drop a file in a well-known directory", and the format is portable across harnesses.
- **Settings as an enforcement surface** — Claude Code's config is a precedence stack (Managed > CLI > local > project > user); most keys override, but **permission rules merge**, and **managed settings cannot be overridden** (a security team denies tools/marketplaces for the whole company). Decision: config is not preference, it is enforcement (connects to ch. 07).

- **Extensibility is also a context budget** — advanced tool use (Anthropic) reframes it: with unlimited tool libraries, the extension must be **loaded on demand**, not registered up front; and plugins toggle on/off to control system-prompt cost. Decision: an extension point that always injects context does not scale — late loading is part of the design (connects to chs. 03 and 05).
- **See also**: the living collection Awesome Harness Engineering — Debugging & Developer Experience gathers more consultable resources for this dimension (patterns, articles, and implementations), curated by problem.

The state of the art

1. Three ecosystem strategies

Round 1's framing persists and got reinforced. **Depth**: hooks reach points the others don't expose — opencode transforms messages and the system prompt before sending, intercepts `permission.ask`, and registers auth providers. **Packaging**: the *extension* as a complete distribution unit (gemini-cli aggregates MCP+commands+hooks+policies into one package; Codex with a manifest + marketplace + App Server JSON-RPC (Remote Procedure Call)). **Interoperability**: adopting the leader's formats instead of inventing your own (OpenHarness with `SKILL.md`/`.claude-plugin`; IronClaw with a compatible `SKILL.md`).

2. The interoperability bet is winning — the "MCP of extensibility"

What in round 1 was the most underrated axis became the dominant trend: **extension formats are converging on standards portable across harnesses**. `SKILL.md`/AgentSkills (OpenClaw uses the `agentskills.io` standard; IronClaw declares compatibility with OpenClaw/Claude), `.claude-plugin` (adopted by OpenHarness), and above all **AGENTS.md** (read by six different harnesses) are doing for extensibility what MCP did for integration. Even the **hook vocabulary** converged — Codex's event set is practically OpenHarness's and Claude Code's (`PreToolUse`/`PostToolUse`/… with Approve/Block/Deny/Ask decisions). Extensibility is ceasing to be a per-harness silo.

3. Marketplaces and security scanning — round 1's gap closed

In round 1, only gemini-cli treated extension code as an attack surface. In round 2 that became the norm, exactly as the plugin *trust triangle* predicted: **OpenClaw** has the **ClawHub** registry with a *trust envelope* + scanning (VirusTotal/ClawScan); Claude Code has a curated official marketplace + community with security triage and **SHA pinning**; **n8n** runs `scan-community-package`; **Goose** checks extensions for malware before loading. Added to the **managed settings** that deny marketplaces enterprise-wide, extension distribution became infrastructure *with containment* — the least-privilege the over-privilege literature demands.

4. Provider-agnosticism became declarative config

The mechanism × policy separation applied to the model: adding a provider stopped being code and became a file. **Goose** has **37 declarative providers via JSON** (an OpenAI-compatible provider = one file); **opencode** has ~26 loaders + hundreds of models via models.dev; **Hermes** has a subclassable `ProviderProfile` (Nous

Portal with 300+ models). The model-agnostic harness — treating the provider as pluggable policy — beat the single-vendor showcase.

5. The next frontier: the harness that extends itself

The embryo of self-extension is already visible: **IronClaw** has **automatic skill extraction** (`learning.rs`) with usage and confidence metrics — the harness observes its own work and writes new skills. It is the bridge to ch. 16 (learning) and to the Voyager/ToolMaker lineage: extensibility that doesn't wait for the user.

EXECUTIVE SUMMARY

What's most modern: format convergence (SKILL.md/.claude-plugin/AGENTS.md as portable standards); marketplaces with security scanning and managed settings; hooks with exit-code as a cross-vendor channel; declarative provider-agnosticism; and the beginning of self-extension. **What to steal:** expose explicit seams (named events, well-known directories) instead of ad-hoc points; adopt portable formats instead of inventing your own; treat third-party extensions as untrusted (scan + least-privilege + managed deny); and make loading late so you don't blow the context.

Hands-on — harness-zero, step 11

Step 11 (`harness-zero/etapas/11-hooks/`) gives harness-zero a **pre/post-tool hook** subsystem: before each tool call, hooks are registered functions (`@hooks.pre_tool`/`@hooks.post_tool`) and the **hook's return value is the control channel** (`"block:reason"` blocks and feeds the reason back to the model; a dict adjusts the arguments) — the completeness exercise proposes the products' external variant: run a user command and read the exit code (0 proceeds; non-zero blocks with the stderr). It is the mechanism (the harness dispatches the hook) separated from the policy (the user decides what the hook does) — the chapter's thesis in ~40 lines. Completeness exercise: you add a `PostToolUse` that runs a linter and returns the errors to the model, and a minimal trust gate (the hook only runs if the directory is trusted).

Check your understanding

1. Why does "open for extension, closed for modification" lead to *hooks* and *plugins* instead of telling the user to fork the harness? (Extension points preserve the core and upgradability; a fork diverges and rots.)
2. You are going to allow a third-party plugin marketplace. Name the central risk (with its name from the literature) and two concrete defenses. (*Trust triangle* / over-privilege; defenses: security scanning + SHA pinning + managed settings that deny + least-privilege.)
3. Your harness needs to support a new model provider without a release. Which design principle makes that "writing a file"? (The mechanism × policy separation — the harness supplies the invocation mechanism, the file supplies the provider's policy.)

Appendix A — How each repository handles extensibility

> Per-harness evidence, with paths — supplemented online, expanded each round.

opencode (round 1) — deep hooks and radical provider agnosticism

Plugins are functions returning `Hooks` (`packages/plugin/`): **~15 points**, including rare ones — transforming messages/system prompt before sending (`experimental.chat.messages.transform`), intercepting `permission.ask`, customizing compaction, and **registering auth providers** (`auth`). Custom tools auto-loaded from `tool/`. And **~26 provider loaders** + hundreds of models via models.dev, on the Vercel AI SDK (Software Development Kit) — the most model-agnostic in production.

gemini-cli (round 1) — the all-in-one package

Extensions (`gemini-extension.json`): an installable package aggregates MCP servers, custom commands, hooks, **permission policies**, skills, and themes. Custom commands in TOML (`FileCommandLoader`). Hooks as a subsystem (`packages/core/src/hooks/`) with a **trust gate** (`trustedHooks.ts` — they only run in trusted folders). Providers: the Google ecosystem.

OpenHarness (round 1) — compatibility as strategy

Markdown skills also loaded from `~/.claude/skills` and `~/.agents/skills` (`SKILL.md` layout); plugins in the `.claude-plugin/plugin.json` format (12 real plugins tested); hooks cover **10 events** with **hot-reload**. Providers as named "workflows" (Anthropic/OpenAI-compatible, Copilot, Kimi, GLM, Ollama…).

Codex CLI (round 2) — full hooks + marketplace + App Server

Full hooks (`hooks/`: PreToolUse/PostToolUse/PreCompact/SessionStart-End/UserPromptSubmit/Stop/SubagentStart-Stop, with Approve/Block/Deny/Ask decisions) and the enterprise knob `allow_managed_hooks_only`; plugins with a manifest and marketplace; skills; configurable providers; profiles; Python/TS SDKs; the **App Server JSON-RPC** as the programmatic backbone.

OpenClaw (round 2) ⭐ — registry with security scanning

Skills in the **AgentSkills** standard (`agentskills.io`) with 6 precedence levels and the public **ClawHub** registry with a *trust envelope* + scanning (VirusTotal/ClawScan); **159 plugins** (tools, channels, providers, hooks, media) with a Plugin SDK; dozens of LLM providers with failover and auth rotation.

IronClaw (round 2) ⭐ — compatible and self-extending

A **`SKILL.md` format compatible** with OpenClaw/Claude; v2 skills with executable snippets, usage/confidence metrics, and **automatic skill extraction** (`learning.rs`); extensions via WASM/MCP/first-party **without restart**; configurable providers (NEAR AI, Gemini OAuth…).

Goose (round 2) — declarative providers and branded distros

Three axes: MCP extensions (6 transport/origin types); recipes/skills; and providers — native + **37 declarative providers via JSON** (adding an OpenAI-compatible provider = creating a file). `CUSTOM_DISTROS.md` (branded distros); `goose-sdk` for embedding; extension malware checks before loading.

Hermes (round 2) — ProviderProfile and plugins

A subclassable `ProviderProfile` (**Nous Portal** with 300+ models under subscription, OpenRouter, your own endpoint); a plugin system (20 directories, a toolset registry, session hooks); Anthropic/Bedrock/Codex/ACP (Agent Client Protocol) adapters.

OpenHands (round 2) — marketplaces and dependency injection

Skill/plugin marketplaces (instance/org/personal); LLM + agent profiles; a pluggable Git-integrations layer; third-party agents via ACP; **sandbox/event-store backends swappable via dependency injection**; litellm for providers.

n8n (round 2) ⭐ — the catalog as extensibility

The strongest point: **the 400+ integration nodes become a tool pool** without writing code (via `usableAsTool` + `$fromAI`); community nodes with a security scanner (`scan-community-package`); ~20 model providers (`LmChat*`).

ohmo (round 2) — extra roots

`~/.ohmo/skills` and `~/.ohmo/plugins` as roots coexisting with the project's; plugins load tools, slash commands, and MCP servers; skills become channel commands; arbitrary per-channel `channel_configs`.

Frameworks (frameworks round)

Frameworks expose extensibility as API: tool registration/`@tool`, lifecycle callbacks/hooks, provider adapters (litellm/model providers), and — increasingly — reading `AGENTS.md`. The portable format (AGENTS.md, SKILL.md) is what brings frameworks and coding harnesses together into a common ecosystem.