12 — Extensibilidade

🕒 estado da arte 2026-07 · revisão 2026-07-26 · 📖 ~12 min de leitura

Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Explicar** por que extensibilidade é "aberto para extensão, fechado para modificação" — extension points em vez de fork;
2. **Distinguir** os quatro eixos de extensão (hooks · comandos/skills · plugins · provedores) e o que cada um resolve;
3. **Comparar** as três estratégias de ecossistema — profundidade, empacotamento, interoperabilidade;
4. **Avaliar** o código de extensão como superfície de ataque (o *trust triangle*) e as defesas (scan, trust envelope, managed settings, least-privilege);
5. **Implementar** um subsistema de hooks pre/post-tool com o retorno do hook como canal de controle no harness-zero (etapa 11).

O problema

Nenhum harness cobre todos os fluxos de trabalho; a extensibilidade decide se o usuário **adapta** o harness ou o **abandona**. Os eixos consagrados:

1. **Hooks** — código do usuário interceptando o ciclo de vida (antes/depois de tool, compactação, sessão).
2. **Skills / comandos custom** — capacidades empacotadas como markdown/config, carregadas sob demanda.
3. **Plugins / extensions** — pacotes distribuíveis agregando tools, comandos, hooks e config.
4. **Provedores de modelo** — a extensão mais estratégica: o harness funciona com qualquer modelo, ou é vitrine de um?

A regra que une os quatro é antiga: **aberto para extensão, fechado para modificação** — o usuário estende sem editar (nem forkar) o core.

Fundamentos científicos

Registro editorial honesto (Princípio I): **não existe canon acadêmico de "extensibilidade de harness de agente"** — é uma lacuna real. As citações duráveis vêm da engenharia de software clássica de arquiteturas extensíveis e da segurança de ecossistemas de plugin, que transferem diretamente.

- **Extension points, não fork** — o princípio aberto-fechado (Meyer, 1988; Martin, 1996) e a arquitetura de plug-ins do Eclipse (Birsan, *ACM Queue* 2005) dão a fundação — e a advertência do "*plug-in hell*":

pontos de extensão mal desenhados viram dívida. Decisão: exponha *seams* explícitos (eventos, diretórios conhecidos), não pontos ad-hoc.

- **Núcleo mínimo, extensões plugáveis** — o padrão Microkernel (Buschmann et al., *POSA* v.1, 1996) e sua encarnação agêntica, AIOS, arXiv 2403.16971 (um kernel que isola escalonamento/memória/tools das aplicações-agente), sustentam a postura "harness como microkernel": um core pequeno que serve de soquete.
- **Mecanismo × política** — Hydra (Levin et al., SOSP '75) é a origem de "separar mecanismo de política". Traduzido: o harness fornece o *mecanismo* (invocar tool, despachar hook, carregar provedor); a *extensão* fornece a política. É por isso que adicionar um provedor de modelo pode ser "escrever um arquivo".
- **Extensão de terceiros não é confiável** — a melhor citação on-topic é LLM (Large Language Model) Platform Security: ChatGPT Plugins, arXiv 2309.10254 (AIES '24): um *trust triangle* plataforma/plugin/usuário com exploits concretos (sequestro de sessão via plugin malicioso). E a base empírica de over-privilege vem da segurança de extensões de browser (Barth et al., NDSS '10: 88% das extensões pedem mais poder do que precisam). Decisão: least-privilege + isolamento + verificação — o mesmo argumento do *tool poisoning* do cap. 06.

(Bibliografia completa e ponteiros: `livro/bibliografia.md`.)

Fontes da indústria

- **Hooks: exit code como canal de controle** — os hooks do Claude Code expõem ~31 eventos de ciclo de vida (`PreToolUse`, `PostToolUse`, `Stop`, `SessionStart`, `UserPromptSubmit`, `PreCompact`, `SubagentStop`…) onde o harness executa comandos do usuário; o **exit code é o canal** (0 = segue / JSON no stdout com allow-deny-ask; 2 = bloqueia com stderr realimentado ao modelo). Decisão: times impõem política (bloquear `rm`, redigir `.env`, auto-lint) de forma **determinística e sem patchar o harness**. E o Codex implementa o mesmo padrão de forma independente (hooks + `allow_managed_hooks_only` para empresas) — hooks são padrão **cross-vendor**, não peculiaridade de um fornecedor.
- **Plugin = unidade de empacotamento; marketplace = catálogo** — o modelo de plugins do Claude Code: um plugin agrega skills, subagentes, hooks, MCP (Model Context Protocol) e LSP (Language Server Protocol) num pacote instalável (`/plugin install nome@marketplace`); um marketplace é um repo git com `.claude-plugin/marketplace.json`. Instala em escopo user/project/local/**managed**, com **pin a SHAs** e um modelo de confiança em dois níveis (marketplace oficial curado + comunidade com triagem de segurança). Decisão: extensão de terceiros vira distribuível **e governável** sem fork.
- **Comandos custom viraram file-drop (e AGENTS.md é o padrão aberto)** — no Claude Code, os comandos slash foram absorvidos pelas skills: largar um arquivo em `.claude/commands/` ou `.claude/skills/` cria o comando, sem registro nem build. E o AGENTS.md virou o formato de config **aberto e multi-tool** — lido por Codex, Cursor, Cline, Windsurf, Gemini CLI e Claude Code. Decisão: o ponto de extensão é "largue um arquivo num diretório conhecido", e o formato é portável entre harnesses.

- **Settings como superfície de enforcement** — a config do Claude Code é uma pilha de precedência (Managed > CLI > local > project > user); a maioria das chaves sobrescreve, mas **regras de permissão fazem merge**, e as **managed settings não podem ser sobrescritas** (uma equipe de segurança nega tools/marketplaces para toda a empresa). Decisão: config não é preferência, é enforcement (liga ao cap. 07).
- **Extensibilidade é também orçamento de contexto** — o advanced tool use (Anthropic) reenquadra: com bibliotecas ilimitadas de tools, a extensão precisa ser **carregada sob demanda**, não registrada de antemão; e plugins se ligam/desligam para controlar o custo de system prompt. Decisão: um ponto de extensão que sempre injeta contexto não escala — o carregamento tardio é parte do design (liga aos caps. 03 e 05).
- **Consulte também**: a coleção viva Awesome Harness Engineering — Debugging & Developer Experience reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

1. Três estratégias de ecossistema

A moldura da rodada 1 persiste e ganhou reforço. **Profundidade**: os hooks alcançam pontos que os outros não expõem — o opencode transforma mensagens e system prompt antes do envio, intercepta `permission.ask` e registra provedores de auth. **Empacotamento**: a *extension* como unidade de distribuição completa (gemini-cli agrega MCP+comandos+hooks+políticas num pacote; Codex com manifest + marketplace + App Server JSON-RPC (Remote Procedure Call)). **Interoperabilidade**: adotar os formatos do líder em vez de inventar os próprios (OpenHarness com `SKILL.md`/`.claude-plugin`; IronClaw com `SKILL.md` compatível).

2. A aposta da interoperabilidade está vencendo — o "MCP da extensibilidade"

O que na rodada 1 era o eixo mais subestimado virou tendência dominante: os **formatos de extensão estão convergindo em padrões portáveis entre harnesses**. `SKILL.md`/AgentSkills (o OpenClaw usa o padrão `agentskills.io`; o IronClaw declara compatibilidade com OpenClaw/Claude), `.claude-plugin` (adotado pelo OpenHarness) e sobretudo o **AGENTS.md** (lido por seis harnesses diferentes) estão fazendo pela extensibilidade o que o MCP fez pela integração. Até o **vocabulário de hooks** convergiu — o conjunto de eventos do Codex é praticamente o do OpenHarness e o do Claude Code (`PreToolUse`/`PostToolUse`/… com decisões Approve/Block/Deny/Ask). A extensibilidade está deixando de ser silo por harness.

3. Marketplaces e scan de segurança — a lacuna da rodada 1 fechou

Na rodada 1, só o gemini-cli tratava código de extensão como superfície de ataque. Na rodada 2 isso virou norma, exatamente como o *trust triangle* de plugins previa: o **OpenClaw** tem o registry **ClawHub** com *trust envelope* + scan (VirusTotal/ClawScan); o Claude Code tem marketplace oficial curado + comunidade com triagem de segurança e **pin a SHA**; o **n8n** roda `scan-community-package`; o **Goose** verifica malware de extensões antes de carregar. Somado às **managed settings** que negam marketplaces enterprise-wide, a

distribuição de extensões virou infraestrutura *com contenção* — o least-privilege que a literatura de over-privilege pede.

4. Provider-agnosticism virou config declarativa

A separação mecanismo × política aplicada ao modelo: adicionar um provedor deixou de ser código e virou arquivo. O **Goose** tem **37 provedores declarativos por JSON** (um provider OpenAI-compatible = um arquivo); o **opencode** tem ~26 loaders + centenas de modelos via models.dev; o **Hermes** tem `ProviderProfile` subclassável (Nous Portal com 300+ modelos). O harness agnóstico de modelo — que trata o provedor como política plugável — venceu a vitrine de um fornecedor só.

5. A próxima fronteira: o harness que se estende sozinho

O embrião da auto-extensão já aparece: o **IronClaw** tem **extração automática de skills** (`learning.rs`) com métricas de uso e confiança — o harness observa o próprio trabalho e escreve skills novas. É a ponte com o cap. 16 (aprendizado) e com a linhagem Voyager/ToolMaker: extensibilidade que não espera o usuário.

> **LEITURA EXECUTIVA**
>
> O que está mais moderno: a convergência de formatos (SKILL.md/.claude-plugin/AGENTS.md como padrões portáveis); marketplaces com scan de segurança e managed settings; hooks com exit-code como canal cross-vendor; provider-agnosticism declarativo; e o começo da auto-extensão. **O que roubar:** exponha seams explícitos (eventos nomeados, diretórios conhecidos) em vez de pontos ad-hoc; adote formatos portáveis em vez de inventar os seus; trate extensão de terceiros como não-confiável (scan + least-privilege + managed deny); e faça o carregamento ser tardio para não estourar o contexto.

Mão na massa — harness-zero, etapa 11

A etapa 11 (`harness-zero/etapas/11-hooks/`) dá ao harness-zero um subsistema de **hooks pre/post-tool**: antes de cada chamada de tool, hooks são funções registradas (`@hooks.pre_tool`/`@hooks.post_tool`) e o **retorno do hook é o canal de controle** (`"block:motivo"` bloqueia e realimenta o motivo ao modelo; um dict ajusta os argumentos) — o exercício de completude propõe a variante externa dos produtos: executar um comando do usuário e ler o exit code (0 segue; não-zero bloqueia com o stderr). É o mecanismo (o harness despacha o hook) separado da política (o usuário decide o que o hook faz) — a tese do capítulo em ~40 linhas. Exercício de completude: você adiciona um `PostToolUse` que roda um linter e devolve os erros ao modelo, e um gate de confiança mínimo (o hook só roda se o diretório for confiável).

Verificação

1. Por que "aberto para extensão, fechado para modificação" leva a *hooks* e *plugins* em vez de instruir o usuário a forkar o harness? (Extension points preservam o core e a atualizabilidade; o fork diverge e

apodrece.)
2. Você vai permitir um marketplace de plugins de terceiros. Cite o risco central (com o nome da literatura) e duas defesas concretas. (*Trust triangle* / over-privilege; defesas: scan de segurança + pin a SHA + managed settings que negam + least-privilege.)
3. Seu harness precisa suportar um novo provedor de modelo sem release. Que princípio de design torna isso "escrever um arquivo"? (Separação mecanismo × política — o harness dá o mecanismo de invocação, o arquivo dá a política do provedor.)

Apêndice A — Como cada repositório trata a extensibilidade

> Evidência por harness, com paths — complementação online, expandida a cada rodada.

opencode (rodada 1) — hooks profundos e agnosticismo radical de provedor

Plugins são funções que retornam `Hooks` (`packages/plugin/`): **~15 pontos**, incluindo raros — transformar mensagens/system prompt antes do envio (`experimental.chat.messages.transform`), interceptar `permission.ask`, customizar compactação e **registrar provedores de auth** (`auth`). Tools custom auto-carregadas de `tool/`. E **~26 loaders de provedor** + centenas de modelos via models.dev, sobre o Vercel AI SDK (Software Development Kit) — o mais agnóstico de modelo em produção.

gemini-cli (rodada 1) — o pacote tudo-em-um

Extensions (`gemini-extension.json`): um pacote instalável agrega MCP servers, comandos custom, hooks, **políticas de permissão**, skills e temas. Comandos custom em TOML (`FileCommandLoader`). Hooks como subsistema (`packages/core/src/hooks/`) com **gate de confiança** (`trustedHooks.ts` — só rodam em pastas confiáveis). Provedores: ecossistema Google.

OpenHarness (rodada 1) — compatibilidade como estratégia

Skills em markdown carregadas também de `~/.claude/skills` e `~/.agents/skills` (layout `SKILL.md`); plugins no formato `.claude-plugin/plugin.json` (12 plugins reais testados); hooks cobrem **10 eventos** com **hot-reload**. Provedores como "workflows" nomeados (Anthropic/OpenAI-compatible, Copilot, Kimi, GLM, Ollama…).

Codex CLI (rodada 2) — hooks completos + marketplace + App Server

Hooks completos (`hooks/`: PreToolUse/PostToolUse/PreCompact/SessionStart-End/UserPromptSubmit/Stop/SubagentStart-Stop, decisões Approve/Block/Deny/Ask) e knob enterprise `allow_managed_hooks_only`; plugins com manifest e marketplace; skills; provedores configuráveis; profiles; SDKs Python/TS; **App Server JSON-RPC** como espinha dorsal programática.

OpenClaw (rodada 2) ⭐ — registry com scan de segurança

Skills no padrão **AgentSkills** (`agentskills.io`) com 6 níveis de precedência e registry público **ClawHub** com *trust envelope* + scan (VirusTotal/ClawScan); **159 plugins** (tools, canais, provedores, hooks, mídia) com Plugin SDK; dezenas de provedores LLM com failover e rotação de auth.

IronClaw (rodada 2) ⭐ — compatível e auto-extensível

Formato **`SKILL.md` compatível** com OpenClaw/Claude; skills v2 com snippets executáveis, métricas de uso/confiança e **extração automática de skills** (`learning.rs`); extensões via WASM/MCP/first-party **sem restart**; providers configuráveis (NEAR AI, Gemini OAuth…).

Goose (rodada 2) — provedores declarativos e distros brandeadas

Três eixos: extensões MCP (6 tipos de transporte/origem); recipes/skills; e provedores — nativos + **37 provedores declarativos por JSON** (adicionar um provider OpenAI-compatible = criar um arquivo). `CUSTOM_DISTROS.md` (distros brandeadas); `goose-sdk` para embutir; verificação de malware de extensões antes do carregamento.

Hermes (rodada 2) — ProviderProfile e plugins

`ProviderProfile` subclassável (**Nous Portal** com 300+ modelos sob assinatura, OpenRouter, endpoint próprio); sistema de plugins (20 diretórios, registry de toolsets, hooks de sessão); adaptadores Anthropic/Bedrock/Codex/ACP (Agent Client Protocol).

OpenHands (rodada 2) — marketplaces e injeção de dependência

Marketplaces de skills/plugins (instance/org/personal); LLM + agent profiles; camada de integrações Git plugável; agentes de terceiros via ACP; **backends de sandbox/event-store trocáveis por injeção de dependências**; litellm para provedores.

n8n (rodada 2) ⭐ — o catálogo como extensibilidade

O ponto mais forte: **os 400+ nós de integração viram pool de tools** sem escrever código (via `usableAsTool` + `$fromAI`); community nodes com scanner de segurança (`scan-community-package`); ~20 providers de modelo (`LmChat*`).

ohmo (rodada 2) — raízes extras

`~/.ohmo/skills` e `~/.ohmo/plugins` como raízes coexistindo com as do projeto; plugins carregam tools, slash commands e servidores MCP; skills viram comandos no canal; `channel_configs` arbitrário por canal.

Frameworks (rodada frameworks)

Os frameworks expõem extensibilidade como API: registro de tools/`@tool`, callbacks/hooks de ciclo de vida, adaptadores de provedor (litellm/model providers), e — cada vez mais — leitura do `AGENTS.md`. O formato portável (AGENTS.md, SKILL.md) é o que aproxima frameworks e harnesses de código num ecossistema comum.