01 — Fundamentos

🕒 estado da arte 2026-07 · revisão 2026-08-01 · 📖 ~13 min de leitura

Este capítulo fixa o vocabulário, a **origem** e o **método** do livro. Antes de comparar harnesses (capítulos 02–13) é preciso responder três perguntas que a primeira edição deixou em aberto: *o que é* um harness, *de onde ele veio* (e o que havia antes), e *com que rigor* este livro o estuda.

1. O que é um harness (definição)

A definição de trabalho vem da lista curada awesome-harness-engineering:

> **Engenharia de harness** é a disciplina de projetar o *scaffolding* — **andaime** ou estrutura de suporte — que envolve um agente de IA (entrega de contexto, interfaces de ferramentas, artefatos de planejamento, loops de verificação, sistemas de memória e sandboxes) e determina se ele tem sucesso ou falha em tarefas reais.

Com o princípio orientador:

> O foco é o *harness*, não o modelo. Cada componente existe porque o modelo não consegue fazê-lo sozinho — e os melhores harnesses são projetados sabendo que esses componentes se tornarão desnecessários conforme os modelos melhoram.

Note o termo central: **scaffolding** (andaime). É a metáfora do livro — a estrutura provisória erguida em volta de algo em construção, que sustenta o trabalho e depois é removida. Guarde a palavra: ela reaparece no subtítulo, no título de cada parte e na §8 (a cláusula de expiração).

> **Para quem está chegando agora — uma imagem que sustenta o livro inteiro.** Pense no modelo como um profissional brilhante no primeiro dia de trabalho numa empresa que ele não conhece: capaz, mas sem mesa, sem acesso aos sistemas, sem saber as regras da casa — e com memória que zera a cada conversa. O harness é tudo que a empresa monta em volta dele: o dossiê do projeto que ele lê ao chegar (contexto, cap. 03), as ferramentas na bancada (cap. 05), o crachá que define onde pode entrar (permissões, cap. 07), o caderno de anotações que sobrevive ao fim do expediente (memória, cap. 08), o supervisor que revisa a entrega antes de ela sair (verificação, cap. 11) — e o expediente em si, o ritmo de trabalhar-conferir-continuar (o loop, cap. 02). Quando os capítulos ficarem técnicos, volte a esta imagem: cada dimensão do livro é uma peça desse escritório.

2. O que havia antes — e por que não eram agentes

"Software que age por você" é uma ideia antiga. As gerações anteriores, porém, resolviam o problema **sem um modelo de linguagem no centro do laço de decisão** — e é isso que as separa de um agente:

- **Sistemas especialistas** (anos 1980): regras `if-then` escritas à mão. Automatizavam decisões, mas não interpretavam objetivos em linguagem natural nem se recuperavam de exceções não previstas.
- **RPA — Robotic Process Automation** (UiPath, Automation Anywhere): robôs que repetem cliques e digitações por *script* fixo. Frágeis a qualquer mudança de tela; sem objetivo, sem recuperação.
- **Chatbots** de intenção (de ELIZA às árvores de diálogo): produziam texto, mas **não executavam ações** no mundo.
- **Assistentes de código como autocomplete**: o **GitHub Copilot** (technical preview em jun/2021), movido pelo modelo **OpenAI Codex** (descendente do GPT-3 ajustado em código), sugeria a próxima linha *dentro do editor* — sem plano, sem ferramentas, sem laço de verificação.

Nenhum deles tinha as **quatro peças** que hoje definem um harness (§4). Faltava-lhes autonomia orientada a objetivos e a capacidade de agir sobre o ambiente **e corrigir o próprio rumo**.

3. Como chegamos aqui — a linhagem técnica

A passagem de "modelo que responde" para "agente que age" foi construída em camadas, cada uma removendo um obstáculo:

1. **Raciocínio explícito.** O *Chain-of-Thought* (Wei et al., 2022) mostrou que pedir ao modelo para "pensar passo a passo" melhora tarefas de raciocínio.
2. **O loop.** O marco decisivo foi **ReAct — *Synergizing Reasoning and Acting in Language Models*** (Yao et al., arXiv:2210.03629, out/2022; ICLR 2023), que intercalou **Pensamento → Ação → Observação**: o modelo raciocina, chama uma ferramenta, observa o resultado e continua. Esse ciclo é o esqueleto de praticamente todo harness moderno (capítulo 02).
3. **A chamada de ferramentas.** Faltava um modo confiável de o modelo *invocar* ferramentas — resolvido quando a OpenAI lançou o **function calling** (jun/2023): o modelo emite JSON estruturado para acionar funções (capítulo 05).
4. **A onda autônoma — e sua lição.** Com raciocínio + ação + ferramentas, veio 2023: **AutoGPT** (Significant Gravitas, mar/2023) e **BabyAGI** (Yohei Nakajima, abr/2023) — loops que se decompunham em subtarefas e se executavam sozinhos. "Falharam" no sentido prático (entravam em círculos, gastavam tokens, perdiam o fio) porque tinham *o loop* mas **não** as outras três peças: gestão de contexto, ferramentas bem projetadas e controle. A lição fundadora da disciplina nasce aí: **o modelo sozinho não basta; o andaime em volta é que decide o sucesso.**
5. **O amadurecimento — os CLIs de código.** As quatro peças foram embutidas em ferramentas de terminal ligadas ao sistema de arquivos e ao Git: **Aider** (Paul Gauthier, abr/2023), **Claude Code** (Anthropic, research preview em fev/2025), **OpenAI Codex CLI** (open source, abr/2025), além de projetos como **Cline**, **OpenHands** e **SWE-agent**.

6. **A padronização.** Com agentes proliferando, vieram os protocolos: o **Model Context Protocol (MCP)**, aberto pela Anthropic (nov/2024), padronizou a conexão a ferramentas e dados (capítulo 06); o **AGENTS.md** consolidou-se como "README para agentes"; o **Agent2Agent (A2A (Agent-to-Agent))** (Google, abr/2025; depois doado à Linux Foundation) endereçou a comunicação *entre* agentes (capítulo 17).

Linha do tempo (marcos): 1980s sistemas especialistas · 2000s–2010s RPA e chatbots · **jun/2021** Copilot (autocomplete) · **out/2022** ReAct · **mar–abr/2023** GPT-4, AutoGPT, BabyAGI, Aider · **jun/2023** function calling · **nov/2024** MCP · **fev/2025** Claude Code · **abr/2025** Codex CLI e A2A.

> **Nota de rigor.** "Codex" designa três coisas distintas — o *modelo* de 2021 (base do Copilot), a *linha de produto* Codex da OpenAI e o *Codex CLI* open source de 2025. O texto as mantém separadas. Datas e fontes desta seção estão na Bibliografia; itens ainda a verificar estão marcados lá.

4. A definição constitutiva: os quatro elementos

A literatura da disciplina converge numa definição do harness como uma **camada de runtime** com quatro elementos necessários e suficientes:

1. **Loop do agente** — o ciclo que alterna entre invocar o modelo e executar o que ele decidiu, até um critério de parada (cap. 02).
2. **Interface de ferramentas** — o contrato pelo qual o modelo age sobre o mundo: ler arquivos, rodar comandos, chamar APIs (cap. 05).
3. **Gestão de contexto** — a montagem, priorização e compressão do que o modelo enxerga a cada chamada (caps. 03–04).
4. **Mecanismos de controle** — permissões, aprovações, sandboxes e limites que restringem o que o agente pode fazer (cap. 07).

Um sistema sem qualquer um dos quatro **não é um harness completo**: um chatbot com ferramentas mas sem loop é um "function caller"; um loop sem controle é um incidente esperando acontecer; ferramentas sem gestão de contexto colapsam em tarefas longas. **Esta é a definição operacional que serve de teste de inclusão** do estudo (§5–6).

As quatro peças numa tarefa real. Peça a um agente: "o teste `test_login` está falhando, corrija". O que acontece, peça a peça: a **gestão de contexto** monta o que o modelo vai enxergar (as regras do projeto, a mensagem, talvez o arquivo do teste); o modelo lê e decide pedir uma ação — "rode o teste e me mostre o erro" — que a **interface de ferramentas** executa de verdade no terminal; o resultado volta, o modelo propõe editar um arquivo, e os **mecanismos de controle** decidem se essa edição pode acontecer direto ou se precisa da sua aprovação; aplicada a edição, o **loop** realimenta o modelo com o novo estado — teste passa? — e repete o ciclo até o critério de parada. Quatro peças, um turno de trabalho. Os capítulos 02–13 são este parágrafo em câmera lenta.

5. De onde vêm os harnesses deste estudo

O corpus é **de código aberto** (o Princípio II do livro: a fonte-base é o código) e se divide em cinco arquétipos — os mesmos do capítulo 00:

- **Harnesses de código** (opencode, gemini-cli, OpenHarness, Codex CLI, Goose, Aider, OpenHands, Grok Build, Pi, Kimi Code, Prime Agent): implementações de referência que juntam as quatro peças num executável.
- **Agentes pessoais self-hosted** (OpenClaw, Hermes Agent, IronClaw, ohmo): o harness a serviço de uma pessoa, com identidade, memória e canais próprios.
- **Agentes organizacionais** (QM): o harness a serviço de uma organização — escopos, permissões por audiência e auditoria como primitivas, com o loop do agente como motor trocável.
- **Harnesses embutidos** (n8n, nó AI Agent): o loop como componente dentro de um produto maior.
- **Frameworks** (LangGraph, CrewAI, OpenAI Agents SDK, Software Agent SDK): expõem loop, estado e ferramentas como primitivas programáveis.

O **teste de inclusão** é a definição da §4: entra quem tem *loop + ferramentas + gestão de contexto + controle*; ficam de fora bibliotecas de modelo puro e meros *wrappers* de uma ferramenta. A lista avaliada, com o repositório e o commit lido de cada um, está no Comparativo e no apêndice do estudo. Recursos consultáveis além do corpus estão na coleção viva Awesome Harness Engineering.

6. O método do estudo (rigor)

Este livro **lê o código-fonte de harnesses reais**, os compara por dimensões e depois **constrói um harness do zero**. Isso não é "opinião de engenheiro": é um desenho de pesquisa híbrido que se apoia em tradições metodológicas consolidadas. Explicitá-las converte o livro de coletânea de impressões em **estudo empírico auditável** — coerente com o Princípio I ("evidência acima de retórica").

Em linguagem simples, antes dos nomes técnicos: o método é (1) escolher sistemas que representem *tipos* diferentes de harness, não os mais famosos; (2) ler o código de cada um seguindo **o mesmo roteiro de perguntas**, anotando o arquivo exato que prova cada resposta; (3) dar notas por uma régua fixa e publicada, para que qualquer pessoa possa discordar olhando a mesma evidência; e (4) construir um harness do zero para testar se os padrões extraídos realmente se sustentam. Os parágrafos a seguir dão os nomes formais de cada uma dessas escolhas e de onde elas vêm — são a genealogia do rigor, e podem ser lidos em diagonal na primeira passada.

Duas fases, dois motores.

- **Fase 1 — descritiva/comparativa:** um **estudo de casos múltiplos** (Yin) apoiado em **Mining Software Repositories** (Hassan, 2008), tratando cada repositório como *dado primário*. A unidade de análise é **o código-fonte**, não o material de marketing nem o comportamento observado em uso.
- **Fase 2 — construtiva/prescritiva:** o `harness-zero` é um exercício de **Design Science Research** (Hevner et al., 2004; processo DSRM de Peffers et al., 2007): projetar e avaliar um artefato que instancia

os princípios extraídos na Fase 1.

Como as dimensões viram medida. As dimensões de comparação descem pelo método **Goal–Question–Metric** (Basili, Caldiera & Rombach): para cada objetivo de harness (contexto, ferramentas, permissões, memória, verificação, loop, orquestração) formulam-se perguntas e, para cada pergunta, **indicadores observáveis no código** (ex.: existe mecanismo de compactação? qual a granularidade do modelo de permissões? há camada de verificação pós-ação?).

Seleção por replicação, não por amostragem. Os casos são escolhidos pela **lógica de replicação** de Yin — *literal* (espera-se o mesmo padrão) ou *teórica* (espera-se diferença previsível) — com critérios explícitos: código aberto e inspecionável na data de corte; pertencer à classe "harness" (§4); relevância de adoção **ou** singularidade arquitetural; diversidade de arquétipos (§5). Para cada caso registram-se **URL, commit/tag e data de leitura**.

Codificação e síntese. A leitura segue um protocolo comum a todos os casos (Runeson & Höst, 2009), combinando codificação indutiva inspirada em *grounded theory* (Stol, Ralph & Fitzgerald, 2016) na descoberta das dimensões e *análise de conteúdo* (Hsieh & Shannon, 2005) com grade fixa na pontuação. A síntese comparativa é uma **feature analysis** no estilo **DESMET** (Kitchenham, Linkman & Law, 1997), na tradição do *benchmarking* como motor de progresso científico (Sim, Easterbrook & Holt, 2003).

Ameaças à validade (taxonomia de Cook & Campbell, 1979, adaptada a estudo de caso):

Tipo	Ameaça	Mitigação declarada
Constructo	as "dimensões" não capturarem o que define um harness	derivação por GQM; definições operacionais publicadas
Interna	atribuir a "boa prática" o que é acaso histórico do projeto	protocolo único; cada afirmação rastreada a trecho/commit
Externa / **obsolescência**	não generalizar; o campo muda em meses	seleção por arquétipos; **data de corte + commits fixos**; a **cláusula de expiração** (§8) é a mitigação declarada, não um enfeite
Conclusão	tratar notas qualitativas como métrica exata	escala e critérios explícitos (DESMET); sem agregação numérica espúria

Assim cada afirmação do livro remete a **um dado no repositório** e a **um procedimento nomeado**. O detalhamento operacional está no Comparativo e no template de avaliação; as referências, na Bibliografia.

7. Taxonomia por problema

Convenção herdada do referencial: organizar a disciplina **pelo problema resolvido, não por fabricante ou modelo**. É a taxonomia que estrutura os capítulos:

Problema	Capítulo
Como o ciclo de decisão-ação funciona e quando para	02 — Loop do Agente
O que o modelo enxerga e como isso é montado	03 — Entrega de Contexto
O que fazer quando a janela de contexto acaba	04 — Compactação
Como o modelo age sobre o mundo	05 — Design de Ferramentas
Como integrar capacidades externas de forma padronizada	06 — MCP
O que o agente pode fazer, e onde	07 — Permissões e Sandboxing
O que persiste entre turnos e entre sessões	08 — Memória e Estado
Como trabalho grande vira passos verificáveis	09 — Planejamento
Como distribuir trabalho entre múltiplos agentes	10 — Subagentes e Orquestração
Como saber se o agente (e o harness) funcionam	11 — Verificação e Evals
Como terceiros estendem o harness	12 — Extensibilidade
Por onde humanos e sistemas usam o agente	13 — Interfaces

8. A cláusula de expiração

A tese mais importante — e menos praticada — da disciplina: **todo componente de harness é uma prótese temporária.** A compactação existe porque janelas de contexto são finitas; o *plan mode* existe porque modelos agem precipitadamente; o *policy engine* existe porque modelos não são confiáveis com comandos destrutivos. Cada premissa tem prazo de validade.

O corolário prático: todo componente deveria documentar **qual melhoria de capacidade do modelo o tornaria desnecessário**. Harnesses que não fazem isso acumulam *scaffolding* morto — complexidade que sobrevive à limitação que a justificava. Como visto na §6, essa cláusula é também a **mitigação declarada** da ameaça de obsolescência: o livro se assume datado. Voltamos a ela no capítulo 14.

9. Artefatos operacionais

A disciplina produziu artefatos-padrão que reaparecem, com variações, em quase todos os harnesses estudados:

- **Arquivo de instruções de projeto** (`AGENTS.md` / `CLAUDE.md` / `GEMINI.md`): regras, convenções e limites que o agente lê antes de qualquer tarefa. Fronteiras claras superam restrições vagas.
- **Artefato de plano** (`PLAN.md`): criado no início da tarefa e atualizado durante a execução, com marcos verificáveis e fronteiras de escopo.
- **Log de implementação** (`IMPLEMENT.md`): registro *append-only* de decisões e desvios do plano.
- **Checklist de harness** (`HARNESS_CHECKLIST.md`): revisão pré-produção cobrindo instruções, ferramentas, contexto, planejamento, permissões e verificação — com a tabela de expiração da §8.

Esses artefatos são o embrião do nosso instrumento de avaliação (ver `benchmark/template/HARNESS_EVAL.md`).

---

*As fontes deste capítulo (históricas e metodológicas) estão consolidadas na Bibliografia, separando as **confirmadas** das que ainda pedem verificação — fiel ao Princípio I.*