05 — Design de Ferramentas

🕒 estado da arte 2026-07 · revisão 2026-07-25 · 📖 ~8 min de leitura

Objetivos de aprendizagem

Ao final deste capítulo, você deve ser capaz de:

1. **Explicar** por que a descrição de uma tool é prompt engineering, não documentação de API;
2. **Derivar** o schema de uma tool a partir de tipos (e justificar por que ninguém mais escreve JSON Schema à mão);
3. **Comparar** os três regimes de escala — catálogo fixo, tool search com carregamento tardio, e code-as-action;
4. **Implementar** a `ToolPort` do harness-zero com schema derivado e erro-como-dado (etapa 2);
5. **Avaliar** quando usar tool calls individuais × código orquestrando tools em sandbox.

O problema

As ferramentas são as "mãos" do agente: o contrato pelo qual o modelo age sobre o mundo. Design de ferramentas é decidir **quais** existem, **como** seus parâmetros são descritos ao modelo, **como** os resultados (e erros) retornam, e **quando** cada uma está disponível. Uma tool mal descrita gera chamadas erradas; um arsenal grande demais dilui a atenção do modelo *e* estoura o orçamento de contexto antes de qualquer trabalho útil; um arsenal pequeno demais força gambiarras via shell.

Fundamentos científicos

- **A evolução do uso de tools** — arXiv 2603.22862 traça a trajetória de single-tool call a orquestração multi-tool, o pano de fundo do "code-as-action".
- **Tool learning como campo** — o survey de tool learning (repo) organiza como agentes aprendem a selecionar e compor ferramentas.

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

Fontes da indústria

- **Writing effective tools for AI agents** (Anthropic Engineering): a fonte canônica — tools são "contratos entre sistemas determinísticos e agentes não-determinísticos"; a descrição é prompt engineering (pequenos refinamentos → grandes ganhos de acerto), o retorno deve ser otimizado por **densidade informacional por token**, e o ciclo é *prototipar → avaliar → colaborar* (o próprio modelo reescreve as tools a partir das transcrições de eval).

- **Code execution with MCP** (Anthropic): carregar todas as definições e passar intermediários pelo contexto é o gargalo — expor cada tool como arquivo TypeScript que o agente orquestra via código levou um caso de **~150.000 → ~2.000 tokens (−98,7%)**.
- **Tool search tool** + **Advanced tool use** (docs + blog): descoberta dinâmica — envie tudo, marque o não-crítico com `defer_loading: true`, o modelo vê só a busca + as essenciais; um setup multi-servidor gasta ~55k tokens de definições antes de trabalhar, e o tool search corta isso em >85%.
- **Programmatic tool calling** (docs): o modelo escreve Python que chama as tools em sandbox e devolve só o destilado — ~38% menos tokens de input num benchmark com 75 tools; 20–40% típico em produção com 10–49 tools.
- **Code Mode** (Cloudflare): a mesma tese, de um fornecedor de infra — o argumento é de *distribuição de treino*: LLMs escrevem código contra APIs conhecidas melhor do que preenchem schemas sintéticos. Convergência de indústria, não peculiaridade de um vendor.
- **Apply Patch** + **GPT-5.1 for developers** (OpenAI): tool de edição **treinada no modelo** (formato V4A de diffs) — explica por que formatos ad-hoc de search/replace perdem para o formato que o modelo viu em treino.
- **Consulte também**: a coleção viva Awesome Harness Engineering — Tool Design reúne mais recursos consultáveis desta dimensão (padrões, artigos e implementações), curados por problema.

O estado da arte

1. O núcleo consensual — e schema derivado de tipos venceu

Os harnesses convergem num núcleo de ~10 tools (ler/escrever/editar arquivo, glob, grep, shell, web fetch/search, todo, delegar) — o kit mínimo de um agente de código. E ninguém escreve JSON Schema à mão: a fonte de verdade é o sistema de tipos (Pydantic no OpenHarness/Hermes, Effect Schema no opencode, classes declarativas no gemini-cli, dataclasses genéricas no software-agent-sdk). O refinamento moderno de qualidade: separar **o que volta ao contexto do modelo do dado estruturado** — o `Observation.to_llm_content` do software-agent-sdk é o design mais limpo (você controla exatamente a densidade informacional que a Anthropic prega).

2. Contexto de tools virou recurso escasso — três regimes de escala

O default de "despejar todas as definições no system prompt" morreu. O estado da arte tem três regimes, e a escolha é por tamanho de catálogo:

- **catálogo fixo** (dezenas de tools): ainda ok mandar tudo;
- **tool search / defer_loading** (centenas de tools, muitos servidores MCP): mantém 3–5 tools quentes, carrega o resto sob demanda — presente como `tool_search`/`tool_discovery` no Codex, Tool Search no OpenClaw, `tool_search` no OpenHarness;
- **code-as-action** (pipelines com dados volumosos): o modelo escreve código que orquestra as tools em sandbox e devolve o destilado — `code-mode` (opencode com V8 embutido, Codex idem), `execute_code` (Hermes chamando tools via RPC (Remote Procedure Call) em "turnos de custo-zero-

contexto"), Code Mode (Goose). A métrica que a indústria passou a reportar não é acurácia isolada, é **acurácia por token de definição**.

3. A interface de edição é treinada, não inventada

A lição mais contraintuitiva: o melhor formato de edição de código não é o que você desenha, é o que o **modelo viu em treino**. Daí o `apply_patch` (V4A) ser tool nativa da OpenAI, o opencode dar `apply_patch` a modelos GPT em vez de `edit`/`write`, e o Aider medir empiricamente qual formato cada modelo aplica bem (`percent_cases_well_formed`). Corolário: a seleção de tools **varia por família de modelo** — reconhecimento explícito de que a interface ideal depende de quem está do outro lado. E erro de tool volta como **dado** (para o modelo se autocorrigir), não como exceção.

> **LEITURA EXECUTIVA**
>
> O que está mais moderno: schema derivado de tipos com separação dado×contexto; os três regimes de escala (fixo → tool search → code-as-action) escolhidos por tamanho de catálogo; e a interface de edição como algo treinado. **O que roubar:** `to_llm_content` (controle de densidade por token); tool search com `defer_loading`; medir o formato de edição por modelo (o `percent_cases_well_formed` do Aider); erro-como-dado.

Mão na massa — harness-zero, etapa 2

A etapa 2 substitui os schemas escritos à mão da etapa 1 por uma `ToolPort`: uma tool é uma função tipada, e o schema é **derivado das anotações** (via `inspect`/`typing`, lendo assinatura e docstring). Você adiciona `read_file` ao lado de `get_time`/`somar`, com erros voltando como texto ao modelo (nunca como exceção que derruba o loop). Exercício de completude: o derivador de schema vem esqueletado para um parâmetro; você estende para tipos compostos.

Verificação

1. Por que a descrição de uma tool é prompt engineering e não documentação de API? (Densidade informacional; iterar sobre transcripts de eval.)
2. Seu agente tem acesso a 8 servidores MCP (200+ tools) e gasta 55k tokens antes de agir. Qual regime de escala você adota, e o que ele carrega quente? (Tool search + defer_loading.)
3. Por que dar `apply_patch` a um modelo pode superar um formato search/replace que você desenhou cuidadosamente? (Distribuição de treino.)

Apêndice A — Como cada repositório trata as ferramentas

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

opencode (rodada 1)

~14 tools + 3 experimentais (`tool/`), Effect Schema, descrições `.txt` separadas; **seleção por modelo** (`registry.ts`: GPT recebe `apply_patch` em vez de `edit`/`write`); ripgrep embutido; experimentais `lsp`, `plan_exit`, `code-mode` (V8).

gemini-cli (rodada 1)

~20–25 tools como classes declarativas (`BaseDeclarativeTool` + `Invocation`), registro filtrado (`maybeRegister`), declarações por família de modelo; shell com processos em background, web search com grounding, tracker opcional (6 tools).

OpenHarness (rodada 1)

43+ tools (`tools/`, `BaseTool` + `input_model` Pydantic → `to_api_schema()`); `is_read_only()` alimenta o paralelismo do loop; multimodal, cron, times, `tool_search`.

Codex CLI (rodada 2)

Crate `tools/` com schemas tipados; `unified_exec` (shell persistente com stdin); **`apply_patch` de primeira classe** (parser streaming + gramática `apply_patch.lark`, variando por modelo); `tool_search`/`tool_discovery`; **code-mode com V8 embutido**.

Goose (rodada 2) ⭐ MCP-nativo

Toda tool é MCP: built-ins de `goose-mcp` são `rmcp::ServerHandler` servidos in-process sobre `DuplexStream`; até developer/shell/edit são "platform extensions" falando `McpClientTrait`.

OpenClaw (rodada 2)

Suíte ampla (`openclaw-tools*.ts`): runtime/files/web/browser CDP/mídia; **Tool Search** e **Code Mode** (JS/TS sobre catálogo oculto); 52 AgentSkills injetadas como bloco compacto, lidas sob demanda.

Hermes (rodada 2)

~40+ tools em **toolsets componíveis** com posturas dinâmicas; `execute_code` (Python chamando tools via RPC, "turnos de custo-zero-contexto"); `schema_sanitizer` por provider.

Aider (rodada 2) ⭐ edit formats

Em vez de tools JSON, **formatos de edição** (`*_coder.py`): whole/diff (SEARCH-REPLACE fuzzy)/udiff/patch; seleção por modelo; **validados por benchmark** (`percent_cases_well_formed`).

software-agent-sdk (rodada frameworks) ⭐ dado×contexto

Contrato Action/Observation/Executor; `Observation.to_llm_content` separa o que volta ao modelo do dado estruturado; toolsets (um `create` → várias tools); anotações MCP-style; `ClientToolSpec` (tool executa na máquina do cliente).

IronClaw (rodada 2)

Tools como **capabilities com descritores tipados** declarando `EffectKind`, credenciais e política de rede; separação visibilidade × autoridade (capability oculta falha fechado); obligations (redação/limites) antes de qualquer efeito.

n8n (rodada 2)

`create-node-as-tool.ts`: **qualquer nó `usableAsTool` vira tool** via `$fromAI('chave','desc',tipo)` → schema Zod derivado; ToolWorkflow (sub-workflow como tool), ToolHttpRequest, ToolCode, ToolThink.

Frameworks (rodada frameworks)

Agents SDK (Software Development Kit): `@function_tool` (Pydantic + griffe com auto-detecção de docstring), 13 tipos incl. hosted; LangGraph: herda `@tool` do langchain-core, adiciona `ToolNode` (execução, injeções); CrewAI: `BaseTool`/`@tool` Pydantic, catálogo `crewai-tools` com 79 diretórios.