05 — Tool Design

🕒 state of the art 2026-07 · revised 2026-07-25 · 📖 ~7 min read

Learning objectives

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

1. **Explain** why a tool's description is prompt engineering, not API documentation;
2. **Derive** a tool's schema from types (and justify why nobody writes JSON Schema by hand anymore);
3. **Compare** the three scaling regimes — fixed catalog, tool search with deferred loading, and code-as-action;
4. **Implement** harness-zero's `ToolPort` with derived schema and error-as-data (step 2);
5. **Evaluate** when to use individual tool calls versus code orchestrating tools in a sandbox.

The problem

Tools are the agent's "hands": the contract through which the model acts on the world. Tool design means deciding **which** tools exist, **how** their parameters are described to the model, **how** results (and errors) come back, and **when** each one is available. A poorly described tool produces wrong calls; an oversized arsenal dilutes the model's attention *and* blows the context budget before any useful work; an undersized arsenal forces workarounds through the shell.

Scientific foundations

- **The evolution of tool use** — arXiv 2603.22862 traces the trajectory from single-tool calls to multi-tool orchestration, the backdrop of "code-as-action".
- **Tool learning as a field** — the tool learning survey (repo) organizes how agents learn to select and compose tools.

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

Industry sources

- **Writing effective tools for AI agents** (Anthropic Engineering): the canonical source — tools are "contracts between deterministic systems and non-deterministic agents"; the description is prompt engineering (small refinements → large accuracy gains), the return value should be optimized for **informational density per token**, and the cycle is *prototype → evaluate → collaborate* (the model itself rewrites the tools from eval transcripts).
- **Code execution with MCP** (Anthropic): loading every definition and passing intermediates through the context is the bottleneck — exposing each tool as a TypeScript file that the agent orchestrates via code

took one case from **~150,000 → ~2,000 tokens (−98.7%)**.

- **Tool search tool** + **Advanced tool use** (docs + blog): dynamic discovery — send everything, mark the non-critical with `defer_loading: true`, the model sees only the search plus the essentials; one multi-server setup spends ~55k tokens on definitions before doing any work, and tool search cuts that by >85%.
- **Programmatic tool calling** (docs): the model writes Python that calls the tools in a sandbox and returns only the distilled result — ~38% fewer input tokens on a 75-tool benchmark; 20–40% typical in production with 10–49 tools.
- **Code Mode** (Cloudflare): the same thesis, from an infrastructure vendor — the argument is about *training distribution*: LLMs write code against known APIs better than they fill in synthetic schemas. Industry convergence, not one vendor's quirk.
- **Apply Patch** + **GPT-5.1 for developers** (OpenAI): an editing tool **trained into the model** (the V4A diff format) — which explains why ad-hoc search/replace formats lose to the format the model saw in training.
- **See also**: the living collection Awesome Harness Engineering — Tool Design gathers more consultable resources for this dimension (patterns, articles, and implementations), curated by problem.

The state of the art

1. The consensual core — and type-derived schema won

The harnesses converge on a core of ~10 tools (read/write/edit file, glob, grep, shell, web fetch/search, todo, delegate) — the minimum kit of a coding agent. And nobody writes JSON Schema by hand: the source of truth is the type system (Pydantic in OpenHarness/Hermes, Effect Schema in opencode, declarative classes in gemini-cli, generic dataclasses in software-agent-sdk). The modern quality refinement: separating **what goes back into the model's context from the structured data** — software-agent-sdk's `Observation.to_llm_content` is the cleanest design (you control exactly the informational density Anthropic preaches).

2. Tool context became a scarce resource — three scaling regimes

The default of "dumping every definition into the system prompt" is dead. The state of the art has three regimes, and the choice is driven by catalog size:

- **fixed catalog** (dozens of tools): still fine to send everything;
- **tool search / defer_loading** (hundreds of tools, many MCP servers): keeps 3–5 tools hot, loads the rest on demand — present as `tool_search`/`tool_discovery` in Codex, Tool Search in OpenClaw, `tool_search` in OpenHarness;
- **code-as-action** (pipelines with bulky data): the model writes code that orchestrates the tools in a sandbox and returns the distilled result — `code-mode` (opencode with embedded V8, Codex likewise), `execute_code` (Hermes calling tools via RPC (Remote Procedure Call) in "zero-context-cost turns"), Code Mode (Goose). The metric the industry now reports is not standalone accuracy, it is **accuracy per definition token**.

3. The editing interface is trained, not invented

The most counterintuitive lesson: the best code-editing format is not the one you design, it is the one the **model saw in training**. Hence `apply_patch` (V4A) being a native OpenAI tool, opencode giving GPT models `apply_patch` instead of `edit`/`write`, and Aider empirically measuring which format each model applies well (`percent_cases_well_formed`). Corollary: tool selection **varies by model family** — an explicit acknowledgment that the ideal interface depends on who is on the other side. And tool errors come back as **data** (so the model can self-correct), not as exceptions.

> **EXECUTIVE SUMMARY**
>
> What is most modern: type-derived schema with data×context separation; the three scaling regimes (fixed → tool search → code-as-action) chosen by catalog size; and the editing interface as something trained. **What to steal:** `to_llm_content` (per-token density control); tool search with `defer_loading`; measuring the editing format per model (Aider's `percent_cases_well_formed`); error-as-data.

Hands-on — harness-zero, step 2

Step 2 replaces step 1's hand-written schemas with a `ToolPort`: a tool is a typed function, and the schema is **derived from the annotations** (via `inspect`/`typing`, reading signature and docstring). You add `read_file` alongside `get_time`/`somar`, with errors returning as text to the model (never as an exception that crashes the loop). Completeness exercise: the schema deriver ships skeletoned for one parameter; you extend it to composite types.

Check your understanding

1. Why is a tool's description prompt engineering and not API documentation? (Informational density; iterating over eval transcripts.)
2. Your agent has access to 8 MCP servers (200+ tools) and spends 55k tokens before acting. Which scaling regime do you adopt, and what does it keep hot? (Tool search + defer_loading.)
3. Why can giving a model `apply_patch` beat a search/replace format you designed carefully? (Training distribution.)

Appendix A — How each repository handles tools

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

opencode (round 1)

~14 tools + 3 experimental (`tool/`), Effect Schema, separate `.txt` descriptions; **per-model selection** (`registry.ts`: GPT gets `apply_patch` instead of `edit`/`write`); embedded ripgrep; experimental `lsp`, `plan_exit`, `code-mode` (V8).

gemini-cli (round 1)

~20–25 tools as declarative classes (`BaseDeclarativeTool` + `Invocation`), filtered registration (`maybeRegister`), declarations per model family; shell with background processes, web search with grounding, optional tracker (6 tools).

OpenHarness (round 1)

43+ tools (`tools/`, `BaseTool` + Pydantic `input_model` → `to_api_schema()`); `is_read_only()` feeds the loop's parallelism; multimodal, cron, teams, `tool_search`.

Codex CLI (round 2)

`tools/` crate with typed schemas; `unified_exec` (persistent shell with stdin); **first-class `apply_patch`** (streaming parser + `apply_patch.lark` grammar, varying by model); `tool_search`/`tool_discovery`; **code-mode with embedded V8**.

Goose (round 2) ⭐ MCP-native

Every tool is MCP: `goose-mcp` built-ins are `rmcp::ServerHandler` served in-process over `DuplexStream`; even developer/shell/edit are "platform extensions" speaking `McpClientTrait`.

OpenClaw (round 2)

Broad suite (`openclaw-tools*.ts`): runtime/files/web/CDP browser/media; **Tool Search** and **Code Mode** (JS/TS over a hidden catalog); 52 AgentSkills injected as a compact block, read on demand.

Hermes (round 2)

~40+ tools in **composable toolsets** with dynamic postures; `execute_code` (Python calling tools via RPC, "zero-context-cost turns"); per-provider `schema_sanitizer`.

Aider (round 2) ⭐ edit formats

Instead of JSON tools, **edit formats** (`*_coder.py`): whole/diff (fuzzy SEARCH-REPLACE)/udiff/patch; per-model selection; **benchmark-validated** (`percent_cases_well_formed`).

software-agent-sdk (frameworks round) ⭐ data×context

Action/Observation/Executor contract; `Observation.to_llm_content` separates what goes back to the model from the structured data; toolsets (one `create` → several tools); MCP-style annotations; `ClientToolSpec` (tool executes on the client's machine).

IronClaw (round 2)

Tools as **capabilities with typed descriptors** declaring `EffectKind`, credentials and network policy; visibility × authority separation (a hidden capability fails closed); obligations (redaction/limits) before any effect.

n8n (round 2)

`create-node-as-tool.ts`: **any `usableAsTool` node becomes a tool** via `$fromAI('chave','desc',tipo)` → derived Zod schema; ToolWorkflow (sub-workflow as tool), ToolHttpRequest, ToolCode, ToolThink.

Frameworks (frameworks round)

Agents SDK (Software Development Kit): `@function_tool` (Pydantic + griffe with docstring auto-detection), 13 types incl. hosted; LangGraph: inherits `@tool` from langchain-core, adds `ToolNode` (execution, injections); CrewAI: Pydantic `BaseTool`/`@tool`, `crewai-tools` catalog with 79 directories.