01 — Foundations

🕒 state of the art 2026-07 · revised 2026-08-01 · 📖 ~13 min read

This chapter pins down the book's vocabulary, **origin** and **method**. Before comparing harnesses (chapters 02–13) we need to answer three questions the first edition left open: *what* a harness is, *where it came from* (and what existed before), and *with what rigor* this book studies it.

1. What a harness is (definition)

The working definition comes from the curated list awesome-harness-engineering:

> **Harness engineering** is the discipline of designing the *scaffolding* — the **support structure** — that surrounds an AI agent (context delivery, tool interfaces, planning artifacts, verification loops, memory systems and sandboxes) and determines whether it succeeds or fails at real tasks.

With the guiding principle:

> The focus is the *harness*, not the model. Each component exists because the model cannot do it on its own — and the best harnesses are designed knowing these components will become unnecessary as models improve.

Note the central term: **scaffolding**. It is the book's metaphor — the temporary structure erected around something under construction, which supports the work and is later removed. Keep the word in mind: it reappears in the subtitle, in the title of each part and in §8 (the expiration clause).

> **If you are just arriving — one image that carries the whole book.** Think of the model as a brilliant professional on their first day at a company they do not know: capable, but with no desk, no access to the systems, no knowledge of the house rules — and with a memory that resets after every conversation. The harness is everything the company builds around them: the project dossier they read on arrival (context, ch. 03), the tools on the bench (ch. 05), the badge that defines where they may enter (permissions, ch. 07), the notebook that survives the end of the shift (memory, ch. 08), the supervisor who reviews the deliverable before it ships (verification, ch. 11) — and the shift itself, the rhythm of work-check-continue (the loop, ch. 02). When the chapters turn technical, come back to this image: every dimension in the book is a piece of that office.

2. What came before — and why they were not agents

"Software that acts on your behalf" is an old idea. The previous generations, however, solved the problem **without a language model at the center of the decision loop** — and that is what separates them from an agent:

- **Expert systems** (1980s): hand-written `if-then` rules. They automated decisions, but neither interpreted goals in natural language nor recovered from unforeseen exceptions.
- **RPA — Robotic Process Automation** (UiPath, Automation Anywhere): bots that replay clicks and keystrokes from a fixed *script*. Fragile to any screen change; no goal, no recovery.
- Intent-based **chatbots** (from ELIZA to dialog trees): they produced text, but **did not execute actions** in the world.
- **Code assistants as autocomplete**: **GitHub Copilot** (technical preview in Jun 2021), powered by the **OpenAI Codex** model (a descendant of GPT-3 fine-tuned on code), suggested the next line *inside the editor* — no plan, no tools, no verification loop.

None of them had the **four pieces** that define a harness today (§4). They lacked goal-oriented autonomy and the ability to act on the environment **and correct their own course**.

3. How we got here — the technical lineage

The passage from "model that answers" to "agent that acts" was built in layers, each removing one obstacle:

1. **Explicit reasoning.** *Chain-of-Thought* (Wei et al., 2022) showed that asking the model to "think step by step" improves reasoning tasks.
2. **The loop.** The decisive milestone was **ReAct — *Synergizing Reasoning and Acting in Language Models*** (Yao et al., arXiv:2210.03629, Oct 2022; ICLR 2023), which interleaved **Thought** → **Action** → **Observation**: the model reasons, calls a tool, observes the result and continues. That cycle is the skeleton of virtually every modern harness (chapter 02).
3. **Tool calling.** What was missing was a reliable way for the model to *invoke* tools — solved when OpenAI shipped **function calling** (Jun 2023): the model emits structured JSON to trigger functions (chapter 05).
4. **The autonomous wave — and its lesson.** With reasoning + action + tools came 2023: **AutoGPT** (Significant Gravitas, Mar 2023) and **BabyAGI** (Yohei Nakajima, Apr 2023) — loops that decomposed themselves into subtasks and ran on their own. They "failed" in the practical sense (going in circles, burning tokens, losing the thread) because they had *the loop* but **not** the other three pieces: context management, well-designed tools and control. The discipline's founding lesson was born there: **the model alone is not enough; the scaffolding around it is what decides success.**
5. **Maturation — the coding CLIs.** The four pieces were embedded into terminal tools wired to the filesystem and to Git: **Aider** (Paul Gauthier, Apr 2023), **Claude Code** (Anthropic, research preview in Feb 2025), **OpenAI Codex CLI** (open source, Apr 2025), plus projects like **Cline**, **OpenHands** and **SWE-agent**.
6. **Standardization.** With agents proliferating came the protocols: the **Model Context Protocol (MCP)**, opened by Anthropic (Nov 2024), standardized the connection to tools and data (chapter 06); **AGENTS.md** consolidated as the "README for agents"; **Agent2Agent (A2A (Agent-to-Agent))**

(Google, Apr 2025; later donated to the Linux Foundation) addressed communication *between* agents (chapter 17).

Timeline (milestones): 1980s expert systems · 2000s–2010s RPA and chatbots · **Jun 2021** Copilot (autocomplete) · **Oct 2022** ReAct · **Mar–Apr 2023** GPT-4, AutoGPT, BabyAGI, Aider · **Jun 2023** function calling · **Nov 2024** MCP · **Feb 2025** Claude Code · **Apr 2025** Codex CLI and A2A.

> **A note on rigor.** "Codex" names three distinct things — the 2021 *model* (the basis of Copilot), OpenAI's Codex *product line* and the open source *Codex CLI* of 2025. The text keeps them separate. Dates and sources for this section are in the Bibliography; items still awaiting verification are flagged there.

4. The constitutive definition: the four elements

The discipline's literature converges on a definition of the harness as a **runtime layer** with four necessary and sufficient elements:

1. **Agent loop** — the cycle that alternates between invoking the model and executing what it decided, until a stopping criterion (ch. 02).
2. **Tool interface** — the contract through which the model acts on the world: reading files, running commands, calling APIs (ch. 05).
3. **Context management** — the assembly, prioritization and compression of what the model sees on each call (chs. 03–04).
4. **Control mechanisms** — permissions, approvals, sandboxes and limits that constrain what the agent can do (ch. 07).

A system missing any of the four **is not a complete harness**: a chatbot with tools but no loop is a "function caller"; a loop without control is an incident waiting to happen; tools without context management collapse on long tasks. **This is the operational definition that serves as the study's inclusion test** (§5–6).

The four pieces in one real task. Ask an agent: "the `test_login` test is failing, fix it". What happens, piece by piece: **context management** assembles what the model will see (the project rules, your message, perhaps the test file); the model reads it and decides to request an action — "run the test and show me the error" — which the **tool interface** actually executes in the terminal; the result comes back, the model proposes editing a file, and the **control mechanisms** decide whether that edit happens directly or needs your approval; once applied, the **loop** feeds the model the new state — does the test pass? — and repeats the cycle until the stop criterion. Four pieces, one turn of work. Chapters 02–13 are this paragraph in slow motion.

5. Where the harnesses in this study come from

The corpus is **open source** (the book's Principle II: the base source is the code) and splits into five archetypes — the same as in chapter 00:

- **Coding harnesses** (opencode, gemini-cli, OpenHarness, Codex CLI, Goose, Aider, OpenHands, Grok Build, Pi, Kimi Code, Prime Agent): reference implementations that put the four pieces together in one executable.
- **Self-hosted personal agents** (OpenClaw, Hermes Agent, IronClaw, ohmo): the harness in the service of one person, with its own identity, memory and channels.
- **Organizational agents** (QM): the harness in the service of an organization — scopes, audience-based permissions and auditing as primitives, with the agent loop as a swappable engine.
- **Embedded harnesses** (n8n, AI Agent node): the loop as a component inside a larger product.
- **Frameworks** (LangGraph, CrewAI, OpenAI Agents SDK, Software Agent SDK): they expose loop, state and tools as programmable primitives.

The **inclusion test** is the definition in §4: whoever has *loop + tools + context management + control* gets in; pure model libraries and mere single-tool *wrappers* stay out. The evaluated list, with each one's repository and the commit read, is in the Comparative and in the study appendix. Resources to consult beyond the corpus are in the living collection Awesome Harness Engineering.

6. The study's method (rigor)

This book **reads the source code of real harnesses**, compares them across dimensions and then **builds a harness from scratch**. That is not "an engineer's opinion": it is a hybrid research design resting on established methodological traditions. Making them explicit turns the book from a collection of impressions into an **auditable empirical study** — consistent with Principle I ("evidence over rhetoric").

In plain language, before the technical names: the method is (1) picking systems that represent different *types* of harness, not the most famous ones; (2) reading each one's code following **the same script of questions**, recording the exact file that proves each answer; (3) scoring against a fixed, published rubric, so anyone can disagree while looking at the same evidence; and (4) building a harness from scratch to test whether the extracted patterns actually hold. The paragraphs that follow give the formal names of each of those choices and where they come from — they are the genealogy of the rigor, and may be skimmed on a first pass.

Two phases, two engines.

- **Phase 1 — descriptive/comparative:** a **multiple-case study** (Yin) supported by **Mining Software Repositories** (Hassan, 2008), treating each repository as *primary data*. The unit of analysis is **the source code**, not marketing material nor behavior observed in use.
- **Phase 2 — constructive/prescriptive:** `harness-zero` is an exercise in **Design Science Research** (Hevner et al., 2004; the DSRM process of Peffers et al., 2007): designing and evaluating an artifact that instantiates the principles extracted in Phase 1.

How dimensions become measures. The comparison dimensions descend via the **Goal–Question–Metric** method (Basili, Caldiera & Rombach): for each harness goal (context, tools, permissions, memory, verification, loop, orchestration) questions are formulated and, for each question, **indicators observable in the code** (e.g.: is

there a compaction mechanism? how granular is the permission model? is there a post-action verification layer?).

Selection by replication, not by sampling. Cases are chosen by Yin's **replication logic** — *literal* (the same pattern is expected) or *theoretical* (a predictable difference is expected) — with explicit criteria: open and inspectable code at the cutoff date; membership in the "harness" class (§4); adoption relevance **or** architectural singularity; archetype diversity (§5). For each case the **URL, commit/tag and reading date** are recorded.

Coding and synthesis. The reading follows a protocol common to all cases (Runeson & Höst, 2009), combining inductive coding inspired by *grounded theory* (Stol, Ralph & Fitzgerald, 2016) in the discovery of the dimensions and *content analysis* (Hsieh & Shannon, 2005) with a fixed grid in the scoring. The comparative synthesis is a **feature analysis** in the **DESMET** style (Kitchenham, Linkman & Law, 1997), in the tradition of *benchmarking* as an engine of scientific progress (Sim, Easterbrook & Holt, 2003).

Threats to validity (Cook & Campbell's 1979 taxonomy, adapted to case study):

Type	Threat	Declared mitigation
Construct	the "dimensions" fail to capture what defines a harness	derivation via GQM; published operational definitions
Internal	attributing to "good practice" what is a project's historical accident	single protocol; every claim traced to a snippet/commit
External / **obsolescence**	failure to generalize; the field changes in months	selection by archetypes; **cutoff date + pinned commits**; the **expiration clause** (§8) is the declared mitigation, not an ornament
Conclusion	treating qualitative scores as exact metrics	explicit scale and criteria (DESMET); no spurious numeric aggregation

Thus every claim in the book traces back to **a datum in the repository** and to **a named procedure**. The operational details are in the Comparative and in the evaluation template; the references, in the Bibliography.

7. Taxonomy by problem

A convention inherited from the reference collection: organize the discipline **by the problem being solved, not by vendor or model**. It is the taxonomy that structures the chapters:

Problem	Chapter
How the decision-action cycle works and when it stops	02 — Agent Loop
What the model sees and how that is assembled	03 — Context Delivery
What to do when the context window runs out	04 — Compaction
How the model acts on the world	05 — Tool Design
How to integrate external capabilities in a standardized way	06 — MCP
What the agent may do, and where	07 — Permissions and Sandboxing
What persists across turns and across sessions	08 — Memory and State
How large work becomes verifiable steps	09 — Planning
How to distribute work across multiple agents	10 — Subagents and Orchestration
How to know whether the agent (and the harness) work	11 — Verification and Evals
How third parties extend the harness	12 — Extensibility
Through what surfaces humans and systems use the agent	13 — Interfaces

8. The expiration clause

The discipline's most important — and least practiced — thesis: **every harness component is a temporary prosthesis.** Compaction exists because context windows are finite; *plan mode* exists because models act rashly; the *policy engine* exists because models are not trustworthy with destructive commands. Each premise has a shelf life.

The practical corollary: every component should document **which model capability improvement would make it unnecessary**. Harnesses that fail to do this accumulate dead *scaffolding* — complexity that outlives the limitation that justified it. As seen in §6, this clause is also the **declared mitigation** of the obsolescence threat: the book assumes it is dated. We return to it in chapter 14.

9. Operational artifacts

The discipline has produced standard artifacts that reappear, with variations, in nearly every harness studied:

- **Project instructions file** (`AGENTS.md` / `CLAUDE.md` / `GEMINI.md`): rules, conventions and limits the agent reads before any task. Clear boundaries beat vague restrictions.
- **Plan artifact** (`PLAN.md`): created at the start of the task and updated during execution, with verifiable milestones and scope boundaries.
- **Implementation log** (`IMPLEMENT.md`): an *append-only* record of decisions and deviations from the plan.

- **Harness checklist** (`HARNESS_CHECKLIST.md`): a pre-production review covering instructions, tools, context, planning, permissions and verification — with the expiration table from §8.

These artifacts are the embryo of our evaluation instrument (see `benchmark/template/HARNESS_EVAL.md`).

---

*This chapter's sources (historical and methodological) are consolidated in the Bibliography, separating the **confirmed** ones from those still awaiting verification — faithful to Principle I.*