arXiv:2609.00006v1 [cs.SE] 15 Jul 2026

Harness Engineering: Anatomy, Architecture, and Evolution of Coding Agents

A Source-Code Study of Eleven Systems

Paul Barbaste*[1,2], Tristan Darrigol[2], Germain Vu[2], and Tom Wiltberger[2]

[1]Inclusive Brains
[2]Wavestone AI Lab

July 2026 ← Актуальный фрез на лето 2026 года

Abstract

An agent is a model plus a harness: the runtime that couples an LLM to the world through a loop, tools, context management, safety controls, orchestration, and extension surfaces. *Harness engineering*, named as a discipline in early 2026, is the design and evolution of that runtime. This paper gives the young discipline its most comprehensive empirical foundation to date. It is a source-code anatomy of eleven production coding harnesses: **Claude Code** (Anthropic), **Codex CLI** (OpenAI), **Gemini CLI** (Google), **Mistral Vibe** (Mistral), **OpenHands**, **Aider**, **Mini-SWE-Agent**, **Hermes** (Nous Research), **Pi**, **OpenCode**, and **OpenClaw**, plus **Omnigent** (Databricks), the first meta-harness we are aware of, analyzed as a contrast point. The paper defines what a harness is, maps its seven canonical subsystems with the minimal and maximal implementation of each, and dissects all eleven systems along those subsystems. It does not benchmark or rank; it describes and compares how the systems are built.

The audit yields 13 cross-cutting observations and a catalog of 29 recurring design patterns. Two absences survive a threefold corpus expansion: across roughly four million lines of Python, TypeScript, and Rust, no agent runtime imports a general-purpose agentic framework (LangChain, LangGraph, AutoGen, or a dozen others; Gemini CLI uses neither of Google's own), and none retrieves code with vector embeddings; the field runs on hand-rolled async loops and deterministic retrieval (`ripgrep`, tree-sitter, glob, auto-discovered Markdown context files). The extensibility standards resolved: SKILL.md skills lead MCP in adoption (9/11 vs. 8/11), with registries, trust tiers, and the corpus's first agent-authored skills; ACP ships in six systems and acquired a third role, *harness hosting*, with OpenHands running Claude Code, Codex, or Gemini CLI as interchangeable backends.

Because the original eight systems were re-pinned rather than replaced, the study also contains a controlled longitudinal sample: the same harnesses, source-diffed across one quarter. The diff shows convergence becoming *imitation* (Codex adopts Claude Code's hook vocabulary verbatim and ships an importer for its sessions and settings; OpenHands reads Claude Code's plugin format), while behavioral policy migrates from prompt prose to configuration, and three of the April edition's observations required substantive revision in place.

These threads converge on the paper's thesis: in the first half of 2026 the coding harness completed a turn from tool to *platform*. Harnesses became importable SDKs while framework vendors shipped harnesses; marketplaces, switching-cost tooling, and enterprise governance layers appeared; the agent became addressable as a model behind an OpenAI-compatible endpoint; and a meta-harness now orchestrates eleven vendor harnesses (half this corpus among them) behind one API, re-implementing the expensive parts and arbitraging the proprietary ones. The paper closes with 18 design recommendations pinned to the observed code, and a 90-line minimum-viable-harness scaffold that implements ten of them.

*Lead and corresponding author.

Keywords: harness engineering, coding agents, LLM agents, agent architecture, platformization, tool use, multi-agent systems, MCP, agent skills

1 Introduction

An agent is a model plus a harness. The model supplies the intelligence; the harness turns that intelligence into work through a loop, tools, context management, safety controls, orchestration, and extension surfaces. In the span of five months, *harness engineering* went from a phrase coined in a vendor blog post to a discipline with practitioner guides, formal definitions, automated-evolution systems, and its own arXiv genealogy (Section 2.1). What the discipline does not yet have is a reference: an account, grounded in production source code, of what a harness actually is, what it is made of, and how the field's leading implementations differ. This paper is that reference, and it arrives with a thesis: the harness has stopped being a tool and become a *platform*. ← сильное утверждение

The existing literature reaches neither goal. Benchmark surveys report scores (typically SWE-Bench [1]) without touching architecture; conceptual taxonomies [5, 6] describe patterns without implementations; the one concurrent source-level taxonomy [34] excludes the provider-native systems that define the field's frontier. None reaches the engineering decisions that determine whether a harness is reliable, extensible, or deployable: how the loop is structured, how tools are defined and sandboxed, how safety is enforced, how context is rationed. Nor does any of it ask where the category itself is heading.

Основные harness тут рассмотрены

We study eleven systems at source level, pinned to July 2026 releases. Four are the flagship products of the major commercial LLM providers: **Claude Code** (Anthropic, TypeScript, via a circulated source snapshot), **Codex CLI** (OpenAI, Rust), **Gemini CLI** (Google, TypeScript), and **Mistral Vibe** (Mistral AI, Python). Seven are open source, spanning the field's full spread: OpenHands (event-sourced SDK, since mid-2026 also a host for rival harnesses), Aider (thirteen polymorphic edit formats), Mini-SWE-Agent (the 100-line research floor), Hermes (the fastest-growing harness on GitHub, with a self-improving skill loop), Pi (the minimal-core countertrend), OpenCode (the most-starred dedicated coding agent, with the corpus's most thoroughgoing client/server architecture), and OpenClaw (a multi-channel gateway, our non-SWE contrast point). A twelfth system, Omnigent (Databricks), is the first *meta*-harness, an orchestration layer over the others, and is analyzed as a second contrast point. Because eight of these systems were already audited in this study's April 2026 edition, their retained snapshots yield a controlled longitudinal sample: the same harnesses, source-diffed across one quarter.

Результаты

The paper makes seven contributions. The first is the discipline's reference anatomy (Section 2): a definition of the harness with its genealogy and boundary cases, and a component map of the seven canonical subsystems, each with its minimal and maximal observed implementation, that organizes the rest of the paper. On that map the paper builds a systematic source-level comparison of eleven harnesses, the first corpus to cover all four provider-native systems, the two most-adopted open-source harnesses, and each system's *signature* capability (Table 4). The audit yields 13 cross-cutting observations and a catalog of 29 recurring patterns, several documented here for the first time: agent-maintained memory pipelines, verify-on-stop guards, lineage compaction, session-tree version control, log-as-queue loops, syntax-aware command permissioning, cache-dialect fanout, and harness mimicry.

У меня была гипотеза про период полураспада harness в полгода

Because the original eight systems were re-pinned rather than replaced, the study also carries a longitudinal analysis (Section 14.5): ninety days of harness evolution measured in source, in which convergence becomes imitation (hook vocabularies copied verbatim, plugin formats adopted, session importers shipped), patterns diffuse down the corpus, and behavioral policy migrates from prompt prose to configuration. Two empirical absences are established at eleven-system scale and survive a threefold corpus expansion: no agent runtime uses a general-purpose agentic framework, and none uses embedding-based retrieval over code; their historical resolution lies in the harness–framework merger. That merger opens onto the platform-turn thesis (Section 14), documented in named artifacts: harness SDKs and framework-built harnesses converging on one shape, plugin and skill

А гипотезы какие?
1) Generic agentic runtime не нужен?
2) Семантический поиск с векторной базой не требуется?

Или гипотез нет и это просто факты?

marketplaces with supply-chain security, cross-vendor session importers, mobile-device-management (MDM) governance, the agent-as-a-model gateway, and a source-level analysis of the first meta-harness. The analysis condenses, finally, into a practitioner's guide of 18 design recommendations, each anchored in a cited observation and a system that implements it, plus a 90-line minimum-viable-harness scaffold that realizes ten of them directly and is compatible with the rest.

The rest of the paper proceeds as follows. Section 2 defines the harness and its seven subsystems. Section 3 surveys related work and the 2026 landscape. Section 4 sets out the methodology; Section 5 gives the system overview. Sections 6–12 dissect the seven subsystems across the corpus. Section 13 ties the cross-cutting observations together. Section 14 develops the platform-turn thesis, including the meta-harness and the longitudinal analysis. Section 15 discusses implications and limits. Section 16 distills the analysis into 18 design recommendations and the 90-line scaffold. Section 17 closes.

2 What Is a Harness?

Before comparing implementations, we fix the object of study. This section defines the harness, traces the short history of the term, separates it from the concepts it is routinely confused with, and lays out the seven subsystems that every harness in our corpus implements in some form. A reader new to the field should leave this section knowing exactly what a harness is and what it is made of; the rest of the paper substantiates each cell of the map against production source code.

2.1 Definition and Genealogy

> *An agent is a model plus a harness. The harness is everything except the model: the runtime that couples an LLM to the world—its loop, its tools, its context, its safety controls, its orchestration, and its extension surfaces. Harness engineering is the discipline of designing and evolving that runtime.*

The one-line algebra **Agent = Model + Harness** was popularized by Trivedy's LangChain engineering series in early 2026 [28], and the term *harness engineering* entered circulation in February 2026: first used in passing by Mitchell Hashimoto, then defined and elaborated by Trivedy in the context of LangChain's Deep Agents work [29].[1] The discipline acquired its practitioner literature within weeks: Böckeler's guide to harness engineering *for coding-agent users* [30] (April), O'Reilly and independent syntheses (June), and community-curated pattern catalogs. An academic wave followed the same arc: Lin et al.'s *Agentic Harness Engineering* [85] automates harness evolution; HARBOR [35] optimizes harnesses against benchmarks; Wang et al. recast code itself as the harness substrate [36]; Rombaut's source-code taxonomy of thirteen open-source scaffolds [34] is the closest methodological relative of the present study (Section 3.3 maps his twelve dimensions onto our seven subsystems); and Macedo [33] gives the concept its first operational definition, deriving necessary and sufficient conditions for what counts as a harness. We adopt Macedo's definitional core—*the layer that wraps a language model and turns it into an agent able to act on a repository*—and extend it with the subsystem decomposition of Section 2.3, which the definitional literature gestures at but has not yet grounded in production source.

2.2 What a Harness Is Not

Usage of the term is loose, and four boundary cases account for most of the confusion [33].

[1] There is a pleasing irony in the provenance: the term was named and defined from within LangChain, the framework vendor whose libraries are absent from every harness runtime in this corpus (Observation 9). LangChain's response to that absence was not to lobby for adoption but to ship a harness of its own—Deep Agents [37], built on LangGraph—which independently converges on the very conventions this corpus documents: `SKILL.md` skills with progressive disclosure, `AGENTS.md` memory, sub-agent spawning, and a todo-planning tool. Section 14.2 returns to this two-way traffic between frameworks and harnesses.

Как код и развернутые сервис

Как набор библиотек и IDE

- **A harness is not a scaffold—quite.** The two words are near-synonyms in practice, and this paper inherits "scaffold" from its own earlier vocabulary. Where a distinction is useful, *scaffold* names the structural code (the loop, the registries) and *harness* names the shipped runtime artifact that embeds it: MINI-SWE-AGENT's scaffold is 100 lines of Python; CLAUDE CODE's harness is a product with a terminal UI, a permission system, and a plugin ecosystem.
- **A harness is not an agentic framework.** A framework (LangChain, AutoGen, CrewAI) is a library the developer imports to *build* an agent; a harness is a runtime the developer works *inside of*. The distinction was clean in 2025 and is dissolving in 2026—from both directions—which is the subject of Section 14.2.
- **A harness is not an evaluation harness.** The SWE-bench "harness" wraps an *agent* to run it against tasks; an agent harness wraps a *model* to make it act. Same word, opposite direction of wrapping.
- **A harness is not an orchestrator.** An orchestrator (or meta-harness, Section 14.4) coordinates one or more harnesses from above and implements no editing loop of its own. OMNIGENT orchestrates eleven vendor harnesses—five of them systems in this paper—and implements no editing loop of its own; it is evidence about harnesses, not one of them.

2.3 The Seven Subsystems

Every system in our corpus, from the 100-line research baseline to the million-line production CLI, must take a position on the same seven subsystems (Figure 1)—even when that position is deliberate absence. They are the analysis dimensions D1–D7 of Section 4, but they are also, we argue, the canonical anatomy of the artifact itself: the minimal implementations show how small each subsystem can shrink—in one case (orchestration) to a deliberate absence—and the maximal implementations show where a decade of engineering headroom lies in each. Table 1 maps the anatomy: what each subsystem does, its minimal and maximal observed forms, and where this paper dissects it.

Классика по мнению авторов.

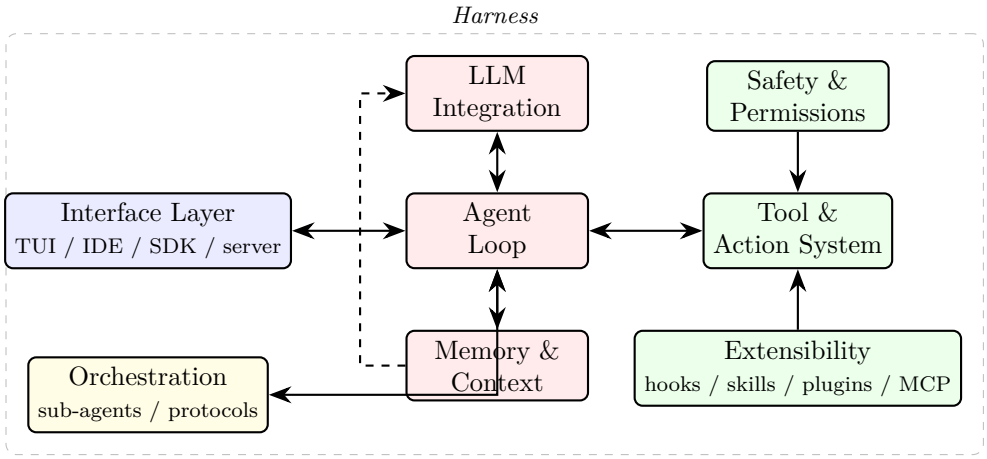


Figure 1: The canonical anatomy of a coding-agent harness: seven subsystems around an agent loop, plus the interface layer through which humans and programs drive it. All eleven systems in the corpus implement these subsystems with varying complexity; Table 1 gives the observed range per subsystem.

B Title. An empirical study of harness design of coding agents" abstract/ introduction/ Tool based agents: 1) plan (task memory) 2) tools 3) context 4) sub agents

Table 1: The component map of a coding-agent harness: the seven subsystems, with the minimal and maximal implementations observed in the corpus.

Subsystem	Role	Minimal form	Maximal form	§
Agent loop	Alternates inference with action execution; owns stop conditions and failure recovery	MINI-SWE-AGENT: a linear `while` over one bash tool	OPENHANDS: event-sourced conversation over a persistent event log, with parallel action batches	6
LLM integration	Speaks provider protocols; assembles the prompt; manages caching, thinking, routing	MINI-SWE-AGENT: one LiteLLM call, one Jinja template	HERMES: five owned transports, 29 provider profiles; CODEX: server-delivered model catalog	7
Tools & actions	Defines and executes what the agent can do, file editing above all	MINI-SWE-AGENT: bash only	CLAUDE CODE: 43 typed tools with deferred loading; CODEX: tool calls as V8-executed code	8
Memory & context	Rations the context window; persists knowledge across turns and sessions	MINI-SWE-AGENT: unbounded linear history	CODEX: agent-maintained cross-session memory pipeline; GEMINI CLI: graph-based context distillation	9
Safety & permissions	Decides what runs, what asks, what is forbidden; isolates execution	MINI-SWE-AGENT: cost and step limits	CODEX: policy rules + LLM approval reviewer + three-platform OS sandbox	10
Orchestration	Spawns and coordinates sub-agents; connects to other agents	AIDER: none (single-agent by design)	CLAUDE CODE: recursive composition; OMNIGENT: cross-vendor coordination (meta-layer)	11
Extensibility	Lets users and ecosystems add capability: config, hooks, skills, plugins, MCP	MINI-SWE-AGENT: structural typing (Python protocols)	PI: everything-is-an-extension runtime; CODEX: marketplace-distributed plugins	12

Two cross-cutting surfaces sit alongside the seven: the *interface layer* (TUI, CLI flags, IDE protocol, HTTP server, SDK) through which humans and programs drive the harness, and the *session substrate* (transcripts, persistence, resume/fork) that several subsystems share. Both recur throughout the corpus analysis, and the interface layer in particular carries the platformization argument of Section 14.

2.4 The Minimal Harness

The decomposition above says what a harness is made of; it does not say how much of each part is needed. The corpus contains an existence proof at the floor: MINI-SWE-AGENT implements all seven subsystems in roughly 100 lines—a `while` loop, one template, one tool, a message list, two limits, no orchestration, and structural typing as its entire extension story—and reports results on SWE-bench Verified in the same range as systems three orders of magnitude larger.[2] Section 16 closes the loop with a 90-line minimum-viable harness that a practitioner can copy and specialize. What separates the floor from the production systems is not task completion but everything else this paper documents: safety, recovery, cost management, extensibility, and—increasingly—the platform

[2]Self-reported figures, different models and dates; see the methodology caveats in Section 15. The point survives the caveats: the floor is low.

surfaces of Section 14.

3 Background and Related Work

Мы его разбирали в серии про SWE Agent

3.1 LLM-Powered Code Agents

The concept of using LLMs as autonomous coding agents gained traction with SWE-Bench [1], a benchmark of real-world GitHub issues requiring end-to-end resolution. SWE-Agent [2] introduced the *Agent-Computer Interface* (ACI), providing LLMs with specialized file viewing and editing tools. The CodeAct paradigm [3] proposed unifying agent actions into executable code, arguing that this subsumes discrete tool calling. ReAct [4] established the reasoning-action loop that underpins most current agent architectures.

A second wave of SWE-specific agents has appeared since. AutoCodeRover [44] replaces text-based search with AST-aware retrieval and spectrum-based fault localization. MASAI [48] decomposes the issue-resolution pipeline into modular sub-agents and reports cost-aware results on SWE-Bench Lite. MAGIS [47] and CodeR [46] both formalize role-based multi-agent coordination for issue resolution (Manager/Custodian/Developer/QA in MAGIS; explicit task graphs in CodeR). Lingma SWE-GPT [49] trains a base model jointly with the SE process loop, an instance of the model–agent co-design we examine in Section 7. SWE-Search [52] layers Monte Carlo tree search over agent trajectories (continuing the line of search-based reasoning developed in Tree of Thoughts [70] and LATS [73], with Plan-and-Solve [71] as the simpler precursor), and Diversity-Empowers-Intelligence [53] ensembles heterogeneous agents under a meta-policy. At the other end of the design spectrum, Agentless [51] reports that a fixed localize/repair/validate pipeline is competitive with agent loops on SWE-Bench Lite, a useful counterweight when discussing whether agentic complexity is necessary. The OpenHands platform paper [45] provides peer-reviewed grounding for the system whose source we audit in Section 6. Concurrent with this study, Lin et al. [85] propose *Agentic Harness Engineering* (AHE), which automatically evolves agent harnesses (tools, middleware, memory, sub-agent configuration) using observability-driven feedback. Starting from a bash-only seed comparable to Mini-SWE-Agent, their system reaches 71.9% on SWE-Bench Verified, and the evolved harness transfers across model families. Their component-level ablation provides quantitative evidence for the architectural emphases we arrive at independently through source-code analysis: tools, middleware, and long-term memory carry the improvement, while the system prompt alone does not.

Кажется, что стоит глянуть.

Эту тоже стоит изучить

3.2 Agent Architecture Surveys

Prior work on agent architectures [5, 6, 66] has identified common patterns including tool use, memory management, and planning. The multi-agent dimension is covered in detail by Guo et al. [67]; the SWE-specific subset is mapped by Liu et al. [68]; AgentBench [69] provides the standard cross-domain evaluation harness. These surveys operate at a conceptual level, describing what components exist rather than how they are implemented. The present work complements them by grounding architectural analysis in concrete implementations, showing that the "same" pattern (e.g., tool use) admits considerable variation in practice.

Тоже заслуживает внимания

3.3 Concurrent Source-Code Taxonomies

The closest methodological relative of this study is Rombaut's *Inside the Scaffold* [34], developed concurrently with our April edition: a source-code taxonomy of thirteen open-source coding-agent scaffolds at pinned commits, organized as twelve dimensions in three layers (control architecture; tool/environment interface; resource management), with the useful finding that most agents *compose* loop primitives (ReAct, generate-test-repair, plan-execute, retry, tree search) rather than implementing a single one. The two studies are complementary rather than redundant. Rombaut's corpus is open-source-only, skews toward benchmark-oriented pipelines (AutoCodeRover, Agentless,

Moatless Tools, DARS, Prometheus, SWE-agent), and explicitly excludes Claude Code for lack of published source. Ours covers all four provider-native harnesses at source level (including the circulated Claude Code snapshot), the gateway and meta-harness contrast points, and the newest mass-adoption systems (Hermes, Pi), and it extends the analysis to dimensions his taxonomy does not reach: permission and safety architecture beyond execution isolation, multi-agent orchestration, extensibility surfaces (hooks, skills, plugins, MCP), prompt content, inter-agent protocols, and the longitudinal and platform analyses of Sections 14.5 and 14. His twelve dimensions map onto our seven subsystems as follows: control-loop strategies, loop driver, and control-flow implementation refine our D1; tool-set design, edit/patch format, and tool-discovery strategy refine D3; context-retrieval paradigm, state management, context compaction, and persistent memory refine D4; execution isolation is the isolation half of our D5; multi-model routing is one cell of our D2. Where his vocabulary is sharper than ours (loop-primitive composition), we adopt it.

3.4 Multi-Agent Systems for Software Engineering

Recent work has explored multi-agent collaboration for software engineering. MetaGPT [7] assigns specialized roles (product manager, architect, engineer) to different agents. ChatDev [50], developed independently, organizes a role-playing development pipeline around chat-chain communication. AutoGen [8] provides a framework for multi-agent conversations. Magentic-One [9] implements an orchestrator pattern with specialized sub-agents. For coding specifically, MAGIS [47] and CodeR [46] pair role decomposition with task graphs, and Chain-of-Agents [72] sequences sub-agents over long inputs. Several systems in the present study incorporate multi-agent patterns, enabling direct comparison of these approaches within a single domain.

3.5 Tool Use in Language Models

Toolformer [10] demonstrated that LLMs can learn to use external tools. Gorilla [11] showed that LLMs can generate accurate API calls when provided with documentation. ToolLLM [58] scaled tool-use evaluation to 16,000+ real-world APIs, and AnyTool [59] demonstrated hierarchical tool routing for large API surfaces. The Model Context Protocol (MCP) [12] proposes a standard interface for connecting LLMs to external tools and data sources. The analysis here finds that MCP has emerged as a cross-system extensibility standard, adopted by eight of the eleven systems studied (all four provider-native agents plus OpenHands, Hermes, OpenCode, and OpenClaw).

3.6 Vendor-Authored Engineering Guidance

A complementary thread to the academic surveys is engineering guidance published directly by major LLM vendors. Between December 2024 and September 2025, Anthropic released a series of four articles that collectively articulate a coherent design philosophy for agents: (i) *Building Effective Agents* [16], which contrasts "workflows" (predefined code paths) with "agents" (LLM-directed dynamic processes), enumerates five composable patterns (prompt chaining, routing, parallelization, orchestrator-workers, evaluator-optimizer), and warns explicitly against general-purpose agentic frameworks; (ii) *Effective Context Engineering for AI Agents* [19], which reframes "prompt engineering" as "context engineering," documents *context rot* in long windows, and recommends a hybrid approach that favors just-in-time (JIT) retrieval via `grep`/`tail`/file-system while acknowledging that pre-indexed retrieval can complement JIT methods for specific use cases; (iii) *Writing Effective Tools for AI Agents* [18], which builds on the agent-computer-interface (ACI) concept introduced in *Building Effective Agents* [16], recommends consolidating tools into "few thoughtful" high-signal operations, and reports SWE-Bench gains from prompt-engineered tool descriptions; (iv) *How We Built Our Multi-Agent Research System* [17], a case study reporting that orchestrator-worker architectures with parallel sub-agents outperformed single-agent baselines by 90.2% on internal evaluation while consuming ∼15× more tokens than a single chat interaction (the comparison is against a simple chat baseline, not against an optimized single-agent pipeline).

Интересная серия

Это конечно ограничивает переносимость результатов

Видимо, стоит разобрать эти 4 статьи отдельно для понимания этого дизайна

The four articles, read as a set, sketch a prescriptive framework that anticipates many of the patterns we later observe across vendors. We use them as a reference baseline against which to evaluate the source-code-level decisions of the studied systems (see especially Section 15.1).

3.7 The 2026 Harness Landscape

The corpus studied in this paper is a deliberate sample from a field that has grown considerably faster than the literature tracking it. A market sweep conducted for this revision (July 2026) identified over two dozen actively maintained coding-agent harnesses beyond the systems analyzed here; Table 2 lists the most significant. Three developments since the first version of this study (April 2026) reshape the context in which the corpus sits.[3]

Every vendor now ships a harness, and the market is consolidating. Beyond the four provider-native systems we analyze in depth, GitHub's Copilot CLI reached general availability in February 2026, Amazon folded its Q Developer CLI into the spec-driven *Kiro* brand, and xAI shipped *Grok Build* [26] in May 2026—notably, a closed-source binary (Rust, per its release artifacts) whose documentation describes wholesale adoption of the open ecosystem's conventions: `AGENTS.md` context files, MCP servers, skills, hooks, and an ACP server mode for editor embedding. Consolidation followed: SpaceX acquired xAI in February 2026 and announced a $60B acquisition of Cursor (Anysphere) in June; Google announced in May that Gemini CLI would transition toward an *Antigravity CLI*, then ended Gemini CLI service for its consumer tiers on June 18 (enterprise Gemini Code Assist licenses retain access and updates). The Google transition is a licensing turn as much as a rebrand: Gemini CLI is Apache-2.0, developed in the open to ∼105k stars, while Antigravity CLI ships as a closed-source Go binary with no published source—so of the three corpus systems whose vendors develop them in the open (Codex, Gemini CLI, Mistral Vibe), one now has a closed successor, and our v0.50.0 pin captures the last major open-source state of a provider-native harness lineage. The harness layer, in other words, is no longer a greenfield: it is strategic infrastructure that vendors buy, rebrand, and gate.

А вот это интересный стратегический вывод. Кажется, что провайдеры хотят продавать не только токены, но и harness as a service

The open-source field has its own centers of gravity. OpenCode [24] is now the most-starred dedicated coding agent on GitHub (∼184k stars), and Hermes [22] the fastest-growing by star-accrual rate (∼212k stars within five months of release; counts as of 2026-07-10, as in Table 2); both enter our corpus in this revision, together with Pi [23], the poster child of the opposite, minimal-core countertrend. Block donated its Rust harness *Goose* to the Linux Foundation's Agentic AI Foundation in April 2026, the first vendor-neutral governance story we identified in the space. A community project, *Claw Code* [27], reimplemented Claude Code's architecture in Python and Rust as a self-described clean-room derivative of the same March 2026 source exposure our Claude Code analysis draws on, reaching ∼100k stars within days—suggesting that harness *architecture* itself, not just model access, commands community interest in its own right. A parallel wave of Chinese-vendor harnesses (Alibaba's Qwen Code, a Gemini CLI fork; Moonshot's Kimi CLI; ByteDance's Trae Agent) ties open-weight model releases to their own CLIs. Meanwhile Aider, one of the field's pioneering systems, entered de-facto community-maintenance mode (last stable release February 2026), with architectural development continuing in a community fork (`aider-ce`).

A meta-layer has appeared. Databricks open-sourced Omnigent [25] in June 2026: an orchestration layer that treats entire harnesses (Claude Code, Codex, OpenCode, Hermes, Pi, Cursor) as interchangeable components behind a common API, adding cross-harness policies, a uniform OS sandbox, and shareable multi-device sessions. Section 14.4 analyzes it as this paper's second contrast point; its existence is itself a datum for the platformization argument of Section 14.1.

[3] Landscape events in this subsection are drawn from vendor announcements, release notes, and repository metadata as of 2026-07-10; unlike the corpus claims, they are not source-verified.

Алехин, что не попали в обзор

Table 2: The broader coding-agent harness landscape, July 2026 (systems not in the study corpus). Star counts are approximate GitHub stars as of 2026-07-10; closed-source systems marked "—".

Harness	Vendor	OSS	Stars	Note
Antigravity CLI	Google	✗	—	Announced May 2026 as Gemini CLI's successor; closed-source Go binary; consumer service cut over June 18
Copilot CLI	GitHub/Microsoft	✗	—	GA Feb. 2026; autopilot + fleet modes; deep GitHub integration
Grok Build	xAI (SpaceX)	✗	—	May 2026; Rust; plan-first loop; ACP server; AGENTS.md/MCP/skills
Cursor CLI	Anysphere	✗	—	IDE-harness parity in the terminal; $60B SpaceX acquisition pending
Kiro CLI	Amazon/AWS	✗	—	ex-Q Developer; spec-driven development identity
Cline	Cline Bot Inc.	✓	∼64k	5M+ IDE installs; plan/act workflow; Roo/Kilo fork family
Goose	Block / Linux Found.	✓	∼29k	Rust; native MCP; donated to Agentic AI Foundation Apr. 2026
Claw Code	Community	✓	∼100k+	Self-described clean-room Python/Rust reimplementation of Claude Code
Crush	Charmbracelet	✓	∼26k	Go-native, LSP-aware TUI; ex-opencode (Go) lineage
Qwen Code	Alibaba	✓	∼26k	Gemini CLI fork for Qwen coder models
Kimi CLI	Moonshot AI	✓	∼9k	Skills, MCP, "agent swarm" parallelism
Trae Agent	ByteDance	✓	∼12k	Research-friendly Python CLI agent
Droid	Factory AI	✗	—	Terminal-Bench SOTA Sept. 2025; enterprise CI focus
Amp	Sourcegraph	✗	—	Ad-supported free tier; deep-mode research workflows
Continue CLI	Continue.dev	✓	∼30k	Multi-surface (IDE+CLI); local/privacy focus

4 Methodology

4.1 System Selection

The study covers eleven harnesses plus one meta-harness contrast point, chosen for their spread along three axes: design philosophy, maturity, and market position (Table 3); Section 3.7 situates the corpus in the broader July-2026 field. The set includes one agent per major commercial LLM provider (Claude Code for Anthropic, Codex for OpenAI, Gemini CLI [21] for Google, Mistral Vibe [20] for Mistral) plus seven open-source projects that occupy different research and production niches—among them the three systems added in this revision: Hermes (the fastest-growing open-source harness), Pi (the minimal-core countertrend), and OpenCode (the most-starred dedicated coding agent, with a distinctive client/server split); the carried-over open-source systems are OpenHands [13], Aider [14], Mini-SWE-Agent [15], and OpenClaw. It spans three programming languages (Python, TypeScript, Rust), three orders of magnitude in code size, and design philosophies that range from a research baseline to fully productized CLI tools. All snapshots were re-pinned to the latest available releases in July 2026 (the April-2026 snapshots of the original systems are retained for longitudinal comparison, and Section 14.5 draws on both).

Table 3: System selection criteria. The top eleven rows are the study corpus; OMNIGENT is analyzed as a meta-harness contrast point only.

System	Provider	Language	Version	Selection Rationale
OPENHANDS	OpenHands (ex-All-Hands-AI)	Python	V1 SDK v1.34.0 (Jul. 2026)	Mature modular framework
AIDER	Community (P. Gauthier)	Python	v0.86.3.dev (May 2026; maintenance mode)	Polymorphic edit strategies
CLAUDE CODE	Anthropic	TypeScript	source snapshot Mar. 2026 (binary 2.1.206, Jul. 2026)	Anthropic's flagship agent
CODEX	OpenAI	Rust	rust-v0.144.1 (Jul. 2026)	OpenAI's flagship agent
GEMINI CLI	Google	TypeScript	v0.50.0 (Jul. 2026)	Google's flagship agent
MISTRAL VIBE	Mistral AI	Python	v2.19.1 (Jul. 2026)	Mistral's flagship agent
MINI-SWE-AGENT	Princeton/Stanford	Python	v2.4.5 (Jul. 2026)	Minimalist research baseline
HERMES	Nous Research	Python	0.18.2 (rel. 2026.7.7.2, Jul. 2026)	Self-improving skills; hybrid personal/SWE harness
PI	earendil-works (M. Zechner)	TypeScript	v0.80.6 (Jul. 2026)	Minimal core, maximal extensibility
OPENCODE	Anomaly (ex-SST)	TypeScript	v1.17.18 (Jul. 2026)	Client/server architecture; most-starred dedicated coding agent
OPENCLAW	Community	TypeScript	v2026.6.11 (Jul. 2026)	Contrast: multi-channel assistant
OMNIGENT	Databricks	Python	v0.4.0 (Jul. 2026)	Contrast: meta-harness above harnesses

OpenClaw as an External Contrast Point. OPENCLAW is an outlier in the set. Its README describes it as a *personal AI assistant gateway*, not a coding agent; it spans 20+ messaging platforms (WhatsApp, Slack, Discord, Signal, iMessage, Matrix, and so on) and ships no native code-editing tool (see Table 7, where its entry is *N/A*). Instead of writing code itself, OPENCLAW *delegates* coding tasks to dedicated SWE agents through plugins (`opencode`, `kimi-coding`, `github-copilot`, `copilot-proxy`). It is included as an external contrast point for three reasons. First, its plugin architecture (260 SDK files, manifest-driven discovery, strict import boundaries) isolates extensibility patterns that are SWE-specific from those that belong to a broader agentic-platform category. Second, it adopts the same cross-cutting standards as SWE-first agents (ACP session spawn, MCP, agentskills.io Skills), providing a non-SWE reference to test convergence claims against. Third, it illustrates the gateway-delegation pattern: a multi-channel assistant that consumes rather than implements coding capabilities. HERMES sits halfway between the two categories: like OPENCLAW it is a multi-channel personal assistant gateway (Telegram, Discord, Slack, WhatsApp, Signal, CLI), but unlike OPENCLAW it ships a full native coding toolset, so we treat it as a corpus member rather than a contrast point and flag the hybrid where it matters. Where adoption counts are sensitive to OPENCLAW's inclusion, both the 11-system count and the 10-coding-harness count are reported. Readers who interpret the convergence claims as strictly about coding agents should use the 10-system figures.

Omnigent as a Meta-Harness Contrast Point. OMNIGENT (Databricks, open-sourced June 2026 [25]) is not a coding harness but an orchestration layer *above* harnesses: it wraps

Claude Code, Codex, Cursor, OpenCode, Hermes, Pi, and custom YAML-defined agents behind a common API, adding cross-harness policies, a uniform OS sandbox, and shareable multi-device sessions. Because it does not implement its own editing loop, scoring it on the seven dimensions alongside the corpus would be a category error; instead, Section 14.4 analyzes it as evidence about which harness capabilities are *migrating up* into a meta-layer.

4.2 Dimensions of Analysis

Each system is analyzed along seven dimensions: **(D1)** Agent loop design; **(D2)** LLM integration and model–agent co-design; **(D3)** Tool and action systems; **(D4)** Memory and context management; **(D5)** Safety and permission models; **(D6)** Multi-agent orchestration; **(D7)** Extensibility mechanisms. All versions are as of July 2026 (exact pins in Table 3); for the eight systems carried over from the April 2026 edition of this study, the April snapshots were retained and diffed against the July ones, and Section 14.5 reports what three months of harness evolution look like at source level. Code-size figures are approximate line counts measured on the pinned snapshots, reported to position systems by order of magnitude; they are not reproducible metrics across counting conventions.

5 System Overview

Table 4 presents a high-level comparison of the eleven systems. The corpus spans three orders of magnitude in code size, three programming languages, and fundamentally different design philosophies; the final column names each system's *signature*—the capability that, as of July 2026, no other system in the corpus implements in the same form.[4]

[4]The April edition of this study tabulated SWE-Bench Verified scores per system. We have removed them from the comparison table: the available figures are self-reported, obtained on different model generations and configurations, and several predate the systems' current defaults. For the record, the spring-2026 self-reported figures were: OpenHands 77.6%, Mini-SWE-Agent 74%+, Claude Code 72.7%, Codex 69.1% (all SWE-Bench Verified); the remaining systems publish no comparable number—Pi explicitly solicits real-world session data "instead of toy benchmarks."

Table 4: High-level comparison of the eleven systems, with each system's signature capability. Claude Code cells here and in the other July-dated tables reflect the March-2026 source snapshot, the newest source available to us; the shipping binary (2.1.206, July 2026) may differ.

System	Lang.	Scale	Provider	Tools	Multi-ag.	Sandbox	UI	Signature
OpenHands	Python	Medium	Multi	25+	Yes	Docker/ Apptainer/ remote	Web+CLI +SDK	Resource-locked parallel tools; hosts rival harnesses as ACP backends
Aider	Python	Small	Multi	13 fmt	No	None	CLI	13 polymorphic edit formats; the corpus's only ranked repo map
Claude Code	TS	Large	Anthropic	43	Yes	Opt-in runtime + worktree	CLI	Deferred tool loading; prompt-cache-sharing forks; the reference others copy
Codex	Rust	Very lg.	OpenAI	25–30	Yes	Native (3 OS)	CLI/TUI	Agent-maintained cross-session memory; tool calls as V8-executed code
Gemini CLI	TS	Very lg.	Google	35+	Yes	Native (3 OS)	CLI/TUI	Model routing as runtime scheduling; the corpus's only A2A server
Mistral Vibe	Python	Small	Mistral+	12+	Yes	Worktree opt.	CLI/TUI	Middleware-pipeline loop; rewind exposed over ACP
Mini-SWE-Agent	Python	Tiny	Multi	1	No	Docker+	CLI	The 100-line floor: all seven subsystems, minimally
Hermes	Python	Very lg.	Multi (owned)	69	Yes	6 pluggable backends	CLI/TUI/ Web+28 ch.	Self-improving skill loop; lineage compaction; verify-on-stop
Pi	TS	Medium	Multi, 35 prov.	7	Ext.	None (by design)	CLI/TUI	Everything-is-an-extension core; session-tree version control
OpenCode	TS	Large	Multi	17	Yes	None (policy)	Client/ server	Client/server harness; syntax-aware command permissioning
OpenClaw	TS	Large	Multi	109+	Yes	None	Multi-ch.	Gateway delegation; Active-Memory sub-agent before each reply

"Scale" is qualitative and reflects order-of-magnitude codebase size in the system's primary source tree: Tiny (∼5 K LoC), Small (∼20–40 K), Medium (∼80–110 K), Large (∼500 K+), Very large (∼600 K+, with Codex at ∼1.1 M lines of Rust in the July snapshot). These figures are not directly comparable across languages and counting conventions; they are reported only to position systems on the simplicity-vs-capability axis.

Observation 1. The eleven systems span three orders of magnitude in code size while targeting similar tasks, yet loop sophistication does not predict benchmark performance. Mini-SWE-Agent's minimal linear loop achieves reported results in the same range as OpenHands's event-sourced conversation engine, despite the latter committing substantially more code to loop orchestration alone. Most of the mass in the production systems addresses concerns orthogonal to task completion: safety, user experience, extensibility—and increasingly *clients and transport*: roughly three-fifths of OpenCode's non-test source is its TUI, web, desktop, and SDK clients rather than the harness proper, and Codex's July tree devotes six-figure line counts to app-server transports, plugins, and a realtime voice layer that no benchmark will ever measure.

6 Agent Loop Design

The agent loop, the central control flow that alternates between LLM inference and action execution, is the architectural backbone of every SWE agent. We identify three distinct paradigms across the studied systems.

6.1 Taxonomy of Agent Loop Paradigms

All eleven systems implement variations of the ReAct [4] pattern, but the implementations differ radically in complexity, concurrency, and state management. We identify three paradigms: *iterative action-observation* (nine systems), *reflection-augmented* (Aider, with Hermes relocating the same function into the loop's stop condition), and *coordinator-worker* (an overlay on the iterative loop, prescriptive in Claude Code and Codex, configuration-gated in Hermes). Rombaut's concurrent taxonomy [34] makes the useful finer-grained point that most production loops *compose* primitives (ReAct, generate-test-repair, plan-execute, retry) rather than implementing one; the plan modes that all four provider-native systems now ship (Section 10) are plan-execute grafted onto ReAct.

The field's vocabulary is still being minted around this subsystem. *Loop engineering* entered circulation in June 2026, between this study's two snapshots: Steinberger (OpenClaw's original author) condensed the shift into a widely circulated dictum—stop prompting coding agents, "design loops that prompt your agents" [31]—and Osmani named and structured the practice days later, giving the loop an anatomy of triggers, topology, verifiers, and stop rules [32]. The coinage is complementary to this section's subject rather than a synonym for it: harness engineering builds the *inner* action–observation cycle dissected below, while loop engineering composes that cycle, from the outside, into self-sustaining outer loops that prompt, verify, and re-run the agent without a human authoring each turn. The seam between the two disciplines is already visible in the corpus as the *outer verification loop* pattern (Table 12): OpenHands's `/goal` endpoint runs an LLM judge over each finished run and either re-prompts or stops, and Hermes's verify-on-stop guard vetoes the inner loop's own exit—harness features whose only purpose is to close an outer loop around the inner one.

6.2 Iterative Action-Observation Loop

The most common pattern follows a cycle of prompt construction, LLM inference, tool execution, and observation collection. Nine systems implement this pattern, each with distinctive characteristics.

OpenHands: Event-Sourced Conversation Engine. OpenHands's loop—since the mid-2026 restructuring, in the `software-agent-sdk` repository rather than the application repo—is an event-sourced conversation engine: a `LocalConversation` drives `Agent.step()`, every event is appended to a persistent `EventLog` (one file per event through a `FileStore`, with flock-based locking and secret redaction via a `SecretRegistry`), and the LLM's view of history is a cached projection of the active branch. Conversation state is a *tree* with a movable head, so replay, fork, and branch navigation are first-class. Each `step()` makes one LLM call but executes *all* tool calls of the response as an action batch—optionally in parallel through a `ParallelToolExecutor` governed by a `tool_concurrency_limit` and a resource-lock manager keyed on each tool's declared resources (files, terminal session, browser), so only same-resource calls serialize (the Claude Code paragraph below and Table 18 contrast this with Claude Code's boolean partitioning). The `StuckDetector` survives the rearchitecture intact: the same five failure scenarios as in the V0 codebase (repeating action-observation pairs, repeating actions with errors, monologue loops, alternating patterns, and context-window error loops), now with configurable thresholds and enabled by default.

Claude Code: Streaming Loop with Concurrent Tool Batching. Claude Code's loop responses stream via Server-Sent Events (SSE) with fine-grained event handling (`content_block_start`, `content_block_delta`, `content_block_stop`). The most distinctive feature is *intelligent tool batching*: tool calls are partitioned into batches by concurrency safety via a single-pass reduce algorithm. Tools default to `isConcurrencySafe = false` (the safe default), and must explicitly opt in to parallel execution. Read-only tools (grep, glob, file read) are safe; write tools (edit, bash) are not. This design enables significant latency reduction when the LLM requests multiple independent reads simultaneously.

Codex: Tokio Async State Machine. CODEX's loop is implemented in Rust as a Tokio-based async state machine. A `Session` struct orchestrates turns via streaming `ResponseItem` events from the OpenAI Responses API (deserialized from SSE or WebSocket stream frames), with tool invocations processed through `FuturesOrdered` for ordered parallel execution, now factored through a dedicated `ToolCallRuntime`. Key entry points: `Codex::spawn()` creates a new session, `submit_with_id()` submits operations, and `next_event()` provides non-blocking event reading for streaming responses (the once-monolithic `codex.rs` has been split into `session/` modules and a `CodexThread` abstraction, but the entry points survive).

Mini-SWE-Agent: Minimal Linear Loop. MINI-SWE-AGENT implements the simplest variant (Listing 1):

```python
def run(self, task):
    self.add_messages(model.format_message(
        "system", render_template(system_template)))
    self.add_messages(model.format_message(
        "user", render_template(instance_template, task=task)))
    while True:
        result = self.step()
        if result.role == "exit":
            return result
def step(self):
    self.query()
    return self.execute_actions()
def query(self):
    self.check_limits()
    self.n_calls += 1
    msg = self.model.query(self.messages)
    self.add_messages(msg)
def execute_actions(self):
    for action in msg["extra"]["actions"]:
        obs = self.env.execute(action)  # subprocess.run()
        self.add_messages(
            model.format_observation_messages(obs))
```

Listing 1: MINI-SWE-AGENT agent loop (condensed; omits format-error handling, cost/time accounting, and trajectory saving present in v2.4.5).

There is no state machine, no event sourcing, and no concurrency mechanism. The message list grows linearly and unboundedly, relying entirely on the LLM's native context window. The minimalism is deliberate: the design intentionally isolates LLM capability from agent scaffolding complexity, to see how much of one can substitute for the other. Even the floor drifts upward, though: the v2.4 line added a wall-clock time limit on runs and a cap on *consecutive malformed-action responses*—the loop now aborts after N successive format errors instead of cycling indefinitely—so the corpus's minimal system has acquired a minimal stuck detector.

Mistral Vibe: Middleware-Pipeline Loop. MISTRAL VIBE's loop is a Python async/await iterative loop in which turn-level policies are factored out into a composable *middleware pipeline*. The core conversation loop drives a standard prompt → LLM → tools cycle, but each iteration first walks a stack of middleware pre-turn checks and then yields events to the caller as they complete. Six middlewares ship out of the box: `TurnLimitMiddleware`, `PriceLimitMiddleware` (cost cap per session), `TokenLimitMiddleware` (session-total token cap, added in the 2.9 line), `AutoCompactMiddleware` (triggers summarization at a token threshold), `ContextWarningMiddleware` (warns when the conversation approaches the context-window limit), and `ReadOnlyAgentMiddleware` (gates write tools for read-only agent profiles); user-cancellation is handled inline via an `is_user_cancellation_event()` check rather than as a middleware. Tool calls within a single LLM response are executed concurrently by spawning each tool as an `asyncio.create_task()` and yielding events as they finish, a finer-grained variant of CODEX's `FuturesOrdered`. The middleware design is unique in the corpus. New turn-level policies can be added without modifying the loop body, and the same loop powers six registered agent profiles (`default`, `plan`, `accept-edits`, `auto-approve`, `explore`, `lean`) simply by swapping the middleware composition; a seventh `chat` profile is not registered for CLI sessions

but is dynamically registered by the ACP layer for IDE integrations. Plan mode has meanwhile become structural rather than prompt-only: a plan file under a dedicated plans directory is the profile's only writable target, and an `exit_plan_mode` tool emits a `PlanReviewRequestedEvent` that gates the transition back to write-capable profiles.

Gemini CLI: Async-Generator Loop with Hybrid Loop Detection. GEMINI CLI's loop is implemented in TypeScript as an async generator. Each iteration yields typed `ServerGeminiStreamEvent` tuples to the UI, runs the active model via a `ModelRouterService`, streams the response via the `@google/genai` SDK, and dispatches function calls through a state-machine `Scheduler` that walks each tool through *Validating → Executing → Completed/Errored.* A distinctive feature is the `LoopDetectionService`, which is hybrid rather than purely deterministic: SHA-256 hashing catches the common failure modes cheaply (five identical tool calls or ten identical content chunks abort the loop), and after thirty turns in a single prompt an LLM-based self-check runs at an adaptive interval—a two-tier design that contrasts with OPENHANDS's five hand-enumerated scenarios. A session turn limit (default 100) provides a hard ceiling, and a set of eleven lifecycle hook events (`BeforeAgent`/`AfterAgent`, `BeforeModel`/`AfterModel`, `BeforeToolSelection`, `PreCompress`, and others) lets extensions inspect or veto the loop. The scheduler executes tool calls in parallel and, since v0.45, partitions by concurrency safety: file-mutating tools (`edit`, `write_file`) and `update_topic` are hard-forced to sequential execution, and—uniquely in the corpus—the model itself is given a concurrency knob, an auto-injected `wait_for_previous` boolean on every tool schema through which it can serialize any call. The scheduler streams live tool output to the UI via an `outputUpdateHandler` callback so the user sees long-running commands progress.

Hermes: Budgeted Loop with Stop-Guards. HERMES runs a single iterative tool-calling loop per user turn, bounded by a thread-safe `IterationBudget` (default 90, with `execute_code` iterations refunded and a final grace call that lets the model summarize when the budget expires). What distinguishes it is not the loop body but its *exits*: eleven enumerated turn-exit reasons, and two stop-guards that can veto a premature final answer. The *verify-on-stop* guard rewrites a text-only response into a continuation whenever the turn mutated code files without producing fresh verification evidence—the reflection loop's function, relocated from inside the loop to its stop condition, at a fraction of the cost. Stuck detection hashes tool name plus sorted-JSON arguments (SHA-256), warning after two identical failures and halting after eight—but the hard stop ships *disabled*: the default posture is warnings appended to tool results, trusting the model to self-correct. Tool batches parallelize on a pool of eight workers only when every call is read-only-safe or path-scoped with non-overlapping prefixes; mid-turn user steering is spliced into the last tool result behind an anti-injection marker.

Pi: Functional Core with Steering Queues. PI's loop is the purest iterative implementation in the corpus after MINI-SWE-AGENT: a ∼790-line functional core with no planner, no reflection step, no turn cap, no stuck detection, and no cost kill-switch—the loop runs until the model stops calling tools, and all of those absences are documented design refusals. Two primitives stand out. Mid-run user input is first-class: a `steer()` queue injects messages after the current turn's tool calls, while `followUp()` drains only when the agent would otherwise stop. And tool calls execute in parallel by default via `Promise.all`, with a realpath-keyed per-file mutation queue serializing edits and a distinctive *truncation-poisoning guard*: when the assistant message was cut off at the length limit, *all* of its tool calls are failed unexecuted, because salvage-parsed streaming arguments can validate while being silently incomplete. Recovery is layered instead of enumerated: a ∼40-pattern transient-error retry allowlist and a one-shot compact-and-retry on context overflow.

OpenCode: Log-as-Queue Loop. OPENCODE's loop is a literal `while(true)` with an unusual twist: the message log doubles as the work queue. Pending sub-agent spawns and compactions are

not control-flow branches but persisted *message parts* that the loop dequeues one per iteration—which makes the loop trivially resumable across process restarts, since the queue *is* the transcript. There is no default step cap (`maxSteps` defaults to infinity); when a configured cap is reached, the harness appends an assistant message declaring tools disabled rather than raising an error. Stuck detection is a single heuristic routed, characteristically, through the permission system: three consecutive byte-identical tool calls raise a `doom_loop` permission *ask* rather than an automated abort—the user, not the harness, decides whether the repetition is a bug. Parallel tool calls execute concurrently with no harness cap, and retry is harness-level with retry-after-aware exponential backoff; context overflow is never retried—it flips to compaction.

6.3 Reflection-Augmented Loop

Aider extends the basic loop with a *reflection* mechanism. The `run_one()` method implements a nested loop: after each LLM response, the system applies edits, then checks for lint errors (via a three-stage Python pipeline: syntax check → compile check → flake8), test failures, and unresolved file mentions. If issues are detected, a `reflected_message` triggers re-invocation of the LLM with corrective context, up to a configurable maximum (default: 3 reflections).

The linter integration is particularly notable: it uses `TreeContext` from `grep_ast` to show code context around error lines, providing the LLM with precise locality information for its corrections. Aider remains the only system whose *main loop* is reflection-shaped, but the corpus has grown three structural cousins: Gemini CLI's edit tool ends in an LLM "edit fixer" subcall that repairs a failed match (Section 8.4), OpenCode feeds LSP diagnostics back into every edit result, and Hermes's verify-on-stop guard performs the reflection check once, at turn exit, instead of per iteration. The pattern is the in-the-wild instantiation of a line of work on language-model self-correction, Reflexion [54], Self-Refine [55], CRITIC [56], and Self-Debug [57], that has otherwise stayed largely in the academic literature. Aider's contribution is the integration: lint and test signals are routed back into the loop as the corrective feedback those papers theorize about. A note on its stop condition: Aider has no tool-call loop at all—a turn ends when the completion stream finishes, and `run_one` exits when no reflection is pending or the cap is reached; it never inspects an `end_turn` finish reason.

6.4 Coordinator-Worker Pattern

Claude Code and Codex support a *coordinator mode* where a parent agent orchestrates multiple worker agents; Hermes gates the same shape behind configuration (an `orchestrator` role plus a spawn-depth setting unlocks nested delegation trees, and a separate Kanban-swarm mode runs planning-root → workers → verifier as subprocesses over a SQLite blackboard). This is an overlay on the iterative loop, adding a hierarchical dispatch layer.

In Claude Code, the coordinator spawns sub-agents via `AgentTool`, each receiving a forked context with isolated `AbortController`, cloned file state cache, and suppressed permission dialogs. Workers communicate results back via `<task-notification>` XML blocks, and the coordinator synthesizes findings before delegating the next phase. The workflow is structured: Research → Synthesis → Implementation → Verification.

In Codex, sub-agents are spawned via `AgentControl::spawn_agent()`, receiving dedicated `ThreadId`s and communicating through typed `Mailbox` channels. Each thread maintains its own history, with `SpawnAgentForkMode` controlling context inheritance (`FullHistory` or `LastNTurns(N)`).

We defer the detailed analysis of multi-agent orchestration to Section 11.

6.5 Comparative Analysis

Figure 2 contrasts the three paradigms.

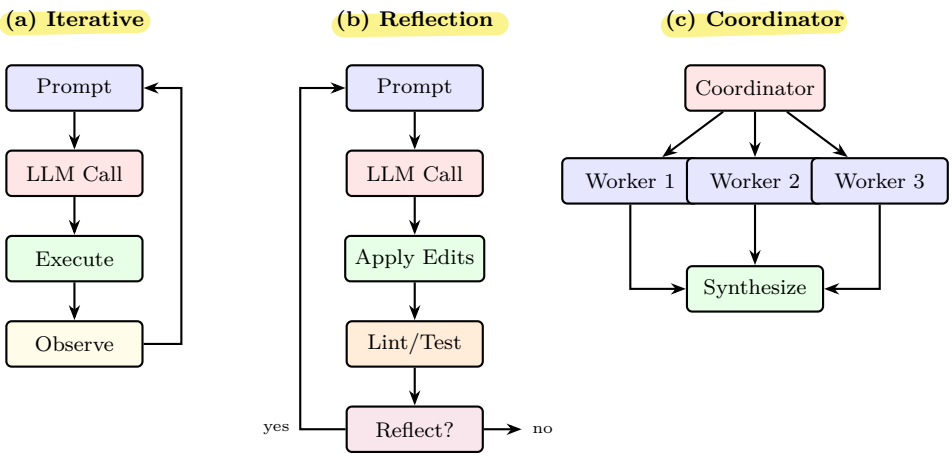


Figure 2: Three agent loop paradigms: (a) iterative action-observation (OpenHands, Claude Code, Codex, Gemini CLI, Mistral Vibe, Mini-SWE-Agent, Hermes, Pi, OpenCode; event-driven variant in OpenClaw), (b) reflection-augmented (Aider), (c) coordinator-worker overlay on (a) (Claude Code, Codex; configuration-gated in Hermes).

Claude и Codex сразу и в варианте „a" и в варианте „b"

Table 5 provides a detailed comparison of loop characteristics.

Table 5: Detailed agent loop characteristics across the eleven systems.

System	Loop Type	Concurrency	Stuck Detect.	Stop Cond.
OpenHands	Event-sourced conversation	Parallel batches (resource locks; limit default 1)	5 scenarios	State=FINISHED
Aider	Generator + reflect	Sequential	Max reflections	stream end, no reflection pending
Claude Code	Streaming + batch	Safety-partitioned	None (manual)	end_turn
Codex	Tokio async SM	FuturesOrdered	None	end_turn
Gemini CLI	Async-gen + scheduler	Parallel; edits forced sequential; model-visible `wait_for_previous`	Hybrid: SHA-256 + LLM check after 30 turns	end_turn / 100
Mistral Vibe	Async + middleware	`asyncio.create_task`	Middleware (turn/price/token/-compact)	end_turn
Mini-SWE-Agent	Linear while	Sequential	step limit; format-error cap; wall clock	exit message
Hermes	Budgeted while + stop-guards	Parallel batches (8 workers, safety-gated)	SHA-256 call signatures (warn-first)	text response unless verify-on-stop vetoes; 90-iter budget
Pi	Functional while + steer/followUp queues	`Promise.all`; per-file mutation queue	None (by design)	end_turn (no cap)
OpenCode	Log-as-queue `while(true)`	Parallel, uncapped	Doom-loop → permission ask	no tool calls; `steps ?? Infinity`
OpenClaw	Event-driven ACP	RPC isolation	Rate limit	Session end

7 LLM Integration and Model–Agent Co-design

How an agent scaffold relates to its underlying LLM is one of the most consequential design decisions in the system. The corpus spans the full range from tight single-provider coupling to complete provider agnosticism, and the position on that spectrum has direct implications for the optimizations available to the scaffold.

7.1 Provider Abstraction Spectrum

The eleven systems adopt five distinct strategies:

1 **Single-provider tight coupling (Anthropic, OpenAI, Google).** Claude Code exclusively uses the Anthropic SDK (`@anthropic-ai/sdk`); Codex uses the OpenAI Responses API with WebSocket streaming; Gemini CLI uses the `@google/genai` SDK and additionally couples to Vertex AI for enterprise deployments. These systems cannot easily switch providers: Claude Code depends on Claude-specific features (extended thinking, prompt caching with static/dynamic boundaries, `tool_use` block format); Codex depends on OpenAI's Responses API wire format and maintains model-specific prompts per GPT generation, delivered as of mid-2026 as *server-side model-catalog data* refreshed at runtime rather than compiled-in templates (Section 7.2); Gemini CLI runs a `ModelRouterService` that, via a chain of pluggable strategies (fallback, override, approval-mode, three classifier strategies—including one that can run against a *local* Gemma model through a managed LiteRT-LM runtime—and default), dispatches each request to a Gemini variant. The router's target set shifted a full model generation between our two snapshots: `gemini-3(.1)-pro-preview` and GA `gemini-3.1-flash-lite` are now the default resolution targets, with Gemma 4 models routable and the 2.5 family surviving only as a no-preview-access fallback—evidence that routing layers exist precisely to absorb model churn so the loop does not.

2 **Provider-first with generic fallback (Mistral).** Mistral Vibe sits between tight coupling and full abstraction: a factory selects between a `MistralBackend` (using the official `mistralai` SDK) and a `GenericBackend` that speaks Anthropic, Vertex, OpenAI-compatible, and—since v2.9—OpenAI Responses endpoints. The Mistral backend exploits Mistral-specific features (the `reasoning_effort` enum mapped from a model's five-level `thinking` field, and the `ThinkChunk` stream block parsed into a separate `reasoning_content` field), while the generic backend provides a portable escape hatch. This is a different philosophy from the LiteLLM-based systems: rather than treating all providers symmetrically, Mistral Vibe privileges its home provider with a deeper integration and treats others as a fallback.

3 **Multi-provider via abstraction layer.** OpenHands, Aider, and Mini-SWE-Agent all use LiteLLM as a universal abstraction, supporting 100+ models across providers. This enables rapid model switching but introduces a dependency on LiteLLM's provider mappings. Aider goes furthest with a model registry containing per-model metadata for 350+ models: edit format, weak model name, cache control, extra API parameters, and reasoning tag handling. OpenHands has meanwhile layered its own routing on top of the abstraction: an `LLMRegistry` keyed by usage, a separate `LLMProfileStore` for named profiles, pluggable `RouterLLM` implementations, and a fallback strategy—so "LiteLLM-based" no longer implies feature-minimal.

4 **Multi-provider via owned transports.** The three systems added in this revision occupy a position the April edition's four-way spectrum did not contain: full multi-provider support with *no* abstraction library, the provider matrix hand-built and exercised to the same depth as the vendor-native integrations. Hermes ships five transport implementations (chat-completions, Anthropic Messages, Bedrock Converse, Codex Responses, and an out-of-process Codex app-server) behind 29 declarative `ProviderProfile` plugins—the profile "is read by the transport instead of receiving 20+ boolean flags"—with model metadata for 3,800+ models pulled from models.dev.

Pi hand-rolls nine wire-protocol implementations across 35 built-in providers, with ∼20 per-model `compat` quirk flags (including ten `thinkingFormat` dialects for OpenAI-compatible hosts) paying the per-provider conditional cost once, centrally—and it exercises the corpus's boldest coupling trick, *harness mimicry*: on a Claude Pro/Max OAuth token it presents Claude Code's identity wholesale (the "You are Claude Code" system-prompt opening, the beta headers, even the canonical tool-name casing, re-cased on the wire) to ride the consumer subscription, and it speaks the ChatGPT-plan Codex backend the same way. OpenCode delegates streaming and tool dispatch to Vercel's AI SDK and the models.dev registry—the only corpus system whose inner LLM plumbing is a third-party SDK—then concentrates its co-design in a ∼1,400-line per-vendor transform matrix (release-date-gated reasoning tiers, per-model temperature defaults, six simultaneous prompt-cache dialects), and can even install *new* providers at runtime from a registry field.

Мимикрия :)

Интересно, что и тут этот слой отдан на откуп внешнему провайдеру

Hybrid plugin-based. OpenClaw implements a pluggable provider system with first-party adapters for 10+ providers, featuring auth profile rotation, last-good tracking for failover, and cooldown expiry for degraded keys.

5

Интересное архитектурное наблюдение

> **Observation 2.** Provider-native optimizations—cache boundaries, extended thinking, reasoning effort, model-specific prompts—are not gated on tight coupling; they are gated on *who pays the per-provider conditional-code cost.* Claude Code splits its prompt at a Blake2b-hashed cache boundary whose static prefix is cached with global scope across sessions (fork-mode sub-agents additionally inherit the parent prompt byte-exactly); Codex keeps per-generation prompts (now delivered as server-side model-catalog data) and a `thread_id`-derived prompt-cache key; Gemini CLI's `ModelRouterService` dispatches each request to the cheapest sufficient Gemini variant. But three multi-provider systems now exercise the same menu from the other end of the spectrum, by paying the conditional-code cost deliberately and centrally: Hermes hand-rolls five transports with provider-agnostic cache markers and bit-perfect prefix normalization that hits even local llama.cpp KV caches; Pi places explicit `cache_control` breakpoints with TTL tiers and audits per-turn cache-miss dollar waste as a first-class metric; OpenCode emits six providers' cache dialects simultaneously and shapes its system prompt to at most two messages (one by default) to match the cache slots. The LiteLLM-based systems (OpenHands, Aider, Mini-SWE-Agent) exercise the menu partially (Table 17), with OpenHands closing the gap through cache-tiered prompt assembly and pluggable routing. What tight coupling still buys, uniquely, is server-side co-evolution: Codex's scaffold behavior—prompts, reasoning tiers, tool modes, even the multi-agent tool generation—is re-tuned per model release from the provider's catalog endpoint without shipping a client. Vendor coupling has become less about *capability* and more about *who controls the update loop*.

7.2 Prompt Engineering Architectures

Prompt construction strategies vary dramatically across the eleven systems. Table 6 summarizes the approaches.

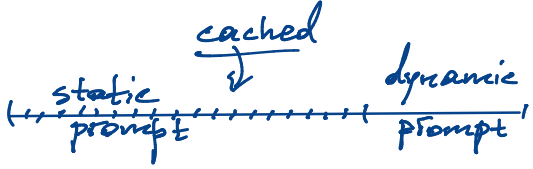
Интересно сравнивать разные подходы

Table 6: Prompt engineering architectures across the eleven systems.

System	Architecture	Sections	Caching	Template
OpenHands	PromptRegistry with cache tiers	18 named static sections (Soul, Role, Security, . . .) + dynamic block	STATIC/DYNAMIC two-block assembly; `cache_control` markers	Python; .j2 escape hatch
Aider	Polymorphic CoderPrompts	Per-edit-format prompts, repo-map, examples	Anthropic markers	Python classes
Claude Code	Modular sections with boundary	12–15 named sections, static/dynamic split	Blake2b hash, boundary marker	TypeScript
Codex	Server-delivered per-model prompts	Capabilities, personality (user-templated), AGENTS.md, planning	`thread_id` cache key	Model-catalog data (bundled fallback)
Gemini CLI	PromptProvider + capability gating	Base, tools, GEMINI.md (global/proj/ext), IDE ctx, memory, skills	None (rebuilt on model change)	TS snippets (modern/legacy)
Mistral Vibe	Per-agent prompts + A/B variants	Base, tools, git status, date, subagent roster, scratchpad, AGENTS.md contents, agent override, skills	Git status cached	Markdown files (GrowthBook-selected variant)
Mini-SWE-Agent	Jinja2 one-shot	system_template, instance_template	Anthropic markers	YAML-embedded
Hermes	Three-tier assembly (stable/context/volatile), built once per session	∼15 guidance blocks, model-family-gated	`system_and_3` markers + bit-perfect prefix normalization	Python constants
Pi	Single ∼170-line builder; tool snippets co-vary with active toolset	Persona, tools, deduplicated guidelines, project context, skills XML, date, cwd	`cache_control` breakpoints + TTL tiers + waste audit	TS concatenation
OpenCode	Model-family prompt matrix (9 base prompts by model-id substring)	Base, env block, AGENTS.md, MCP instructions, skills catalog	Six-dialect breakpoints; `promptCacheKey`=sessionID; system capped at 2 messages	.txt files
OpenClaw	ACP translator	Session config, tool list, thinking level	Per-provider	Runtime config

Gemini CLI: Capability-Gated Snippets with Hierarchical Memory. Gemini CLI's `PromptProvider` assembles the system prompt from base instructions chosen by model capability (`isModernModel ? snippets : snippets.legacy`), tool/function declarations from the registry, and a hierarchically merged set of `GEMINI.md` files discovered at three scopes (global `~/.gemini/`, extension-provided, project-local `.gemini/`). IDE diagnostics and open-file diffs are injected mid-conversation via the IDE companion package; persistent facts are written by the model editing `GEMINI.md` or the private per-project `MEMORY.md` index directly with ordinary edit tools—the system prompt states that no `save_memory` tool exists (Section 9.6). Unlike in Claude Code, the prompt is rebuilt on each model change but not split by a static/dynamic cache boundary—Gemini does not currently expose request-level prompt caching, so cache amortization is left to the API server.

Mistral Vibe: Per-Agent Markdown Prompts. Mistral Vibe's prompts live as plain Markdown files; only the `explore` and `lean` profiles ship a dedicated system prompt, while `plan`, `accept-edits`, and `auto-approve` reuse the default `cli.md` via overrides. A universal system-prompt builder composes a base prompt with auto-generated tool descriptions, a Git context block (output of `git log -oneline -N -decorate` and `git status -porcelain`, cached), and an agent-specific overlay, so the `plan` and `explore` subagents receive entirely different system prompts from the default agent without any code branching. Skill descriptions are appended dynamically. This per-agent specialization is more granular than Claude Code's shared prompt with sub-agent prompt overrides, and structurally similar to Codex's model-specific templates, except keyed on *agent profile* rather than on model generation.



Claude Code: Modular Assembly with Cache Boundary. Claude Code's prompt is assembled from a dozen-plus named sections, divided by a `SYSTEM_PROMPT_DYNAMIC_BOUNDARY` marker. Sections before the boundary (identity, system rules, task guidance, tool patterns, tone) are *static* and cached via Blake2b hashing with `scope: 'global'`. Sections after the boundary (session guidance, memory, environment, MCP instructions, language, scratchpad) are *dynamic* and computed per-turn via `systemPromptSection()` with memoization. This design minimizes prompt cache invalidation: the static prefix (which constitutes the majority of tokens) achieves high cache hit rates across turns and even across sub-agents (via `renderedSystemPrompt` sharing at fork time).

Codex: Per-Model Prompts as Server-Delivered Data. Codex maintains separate prompts per model generation, and the delivery mechanism has itself co-evolved: the compiled-in Markdown templates of early versions still sit in the tree but are no longer referenced by code—the operative prompts ship as `base_instructions` fields of a model manifest (`models.json`, bundled as fallback and refreshed from a remote `/models` endpoint with ETag caching). The July manifest spans GPT-5.2 through a GPT-5.6 family, and the prompts are *personality-templated*: a `{{ personality }}` variable is filled with user-selectable variants (friendly/pragmatic). Prompt content drifts measurably across generations: GPT-5.2's prompt opens with version identity ("You are GPT-5.2 running in the Codex CLI...") and carries explicit no-commit and citation-ban rules; the GPT-5.5 prompt opens "You are Codex, a coding agent based on GPT-5..." (further shortened by the 5.6 family to "You are Codex, an agent based on GPT-5"), expands a rich Personality section, and *drops* the no-commit rule and the anti-gold-plating directives entirely. Behavioral policy is migrating from prompt prose to feature flags: a `codex_git_commit` flag now governs commit behavior. Section 7.3 returns to this thinning.

OpenHands: Prompt Registry with Cache Tiers. OpenHands replaced its Jinja2 `PromptManager` with a `PromptRegistry` of guarded, ordered `PromptSection` objects bucketed into `CacheTier.STATIC` and `DYNAMIC`, rendered as a two-block system message whose static prefix stays byte-stable for provider prompt caching—the same static/dynamic cache-boundary design as Claude Code's, arrived at independently from within a LiteLLM-based multi-provider system. Eighteen named static sections ship (Soul, Role, Memory, Security, SecurityRiskAssessment, VersionControl, and others), with Jinja templates surviving only as an escape hatch.

Hermes: Three-Tier Assembly with Bit-Perfect Prefixes. Hermes builds its prompt once per session in three tiers—stable, context, volatile—with ∼15 guidance blocks gated by model family (tool-use enforcement blocks are sent to GPT/Codex/Gemini/Grok/Qwen/DeepSeek models and never to Claude, with code comments citing observed per-model failures). Cache economics drive the details: a single `system_and_3` strategy places four `cache_control` breakpoints, and the prompt prefix is *bit-perfect normalized* (sorted compact re-serialization of tool-call JSON, a date-only timestamp, a frozen memory snapshot) so that even local llama.cpp and vLLM KV caches hit across turns.

Pi: One Small Builder, Co-Varying with the Toolset. Pi's entire system prompt is a ∼170-line builder emitting 30–40 lines: persona, tool list, deduplicated per-tool guideline snippets (each tool contributes `promptSnippet`/`promptGuidelines`, so the prompt co-varies with the active toolset), `<project_context>` from AGENTS.md, an `<available_skills>` XML index, date and cwd. Almost all behavioral policy is deliberately absent, delegated to user-supplied context files and extensions.

OpenCode: A Model-Family Prompt Matrix. OpenCode dispatches one of nine base prompts by model-id substring—`claude`, `gpt-`, `gemini`, and others each get dedicated .txt prompts with different registers (the Claude prompt opens "You are OpenCode, the best coding agent on the planet.")—generalizing Codex's per-generation templates across vendors. Assembly deliberately

Скромно и со вкусом :)

caps the system prompt at two messages (one by default; plugin-expanded prompts are collapsed back), matching the two leading cache-breakpoint slots it emits in six provider dialects simultaneously.

Aider: Polymorphic Prompts. AIDER's prompt system is unique: each of its 13 registered edit formats (`architect`, `ask`, `context`, `diff`, `diff-fenced`, `editor-diff`, `editor-diff-fenced`, `editor-whole`, `help`, `patch`, `udiff`, `udiff-simple`, `whole`) has a dedicated `CoderPrompts` subclass defining system prompts, examples, and reminders tailored to that format.[5] A factory pattern (`Coder.create()`) selects the format based on the model's metadata. The prompt dynamically includes: file contents, a token-constrained repository map (generated via tree-sitter symbol extraction), chat history, and format-specific editing instructions.

Mini-SWE-Agent: One-Shot Templates. MINI-SWE-AGENT uses Jinja2 templates embedded in YAML configuration files, rendered once at session start. Template variables include system information (`platform.uname()`), model statistics (`n_model_calls`, `model_cost`), and the task description. A notable detail: the default template includes OS-specific instructions (e.g., `sed -i "` for macOS).

7.3 Prompt Content and Rhetorical Style

The previous subsection examined *how* prompts are assembled (cache boundaries, templates, hierarchical merges). We now turn to *what they actually say*: the persona each scaffold projects, the explicit directives it issues to the model, and the rhetorical devices it uses to make those directives stick. We read the canonical system prompt of each system and extracted recurring axes. Several patterns prove convergent across otherwise unrelated codebases.

Identity and persona. Persona openings range from terse to elaborate. MINI-SWE-AGENT is the most minimal: "You are a helpful assistant that can interact with a computer"—though PI's *inner* agent library beats it ("You are a helpful assistant."), with the coding layer adding only "You are an expert coding assistant operating inside pi, a coding agent harness." OPENHANDS adds a role: "You are OpenHands agent, a helpful AI assistant that can interact with a computer to solve tasks"—now wrapped in a `<SOUL>` block loaded from a user-overridable `SOUL.md`. CODEX's GPT-5.2 prompt is the most explicit about model coupling: "You are GPT-5.2 running in the Codex CLI, a terminal-based coding assistant"—an opening the GPT-5.5 generation drops in favor of "You are Codex, a coding agent based on GPT-5..." (the 5.6 family shortens further to "an agent based on GPT-5") with an elaborate Personality section. OPENCODE varies persona *by model*: Claude, Codex, and Meta models are told "You are OpenCode, the best coding agent on the planet," while the default lineage gets the sober "an interactive CLI tool that helps users with software engineering tasks." HERMES layers a general identity ("You are Hermes Agent... created by Nous Research... prioritize being genuinely useful over being verbose") with a coding posture injected only when auto-detected: "Operate like a careful senior engineer." MISTRAL VIBE, which in its April version opened with a complaint about prior behavior ("`CRITICAL:` Users complain you are too verbose"), now opens with a formal *instruction-hierarchy contract*: seven precedence levels (critical > user > repo AGENTS.md > user AGENTS.md > prompt defaults > skills/MCP > external-data-as-data)—prompt-injection defense expressed as a ranking rather than prohibitions. AIDER eschews persona entirely in favor of a role assertion: "Act as an expert software developer. Always use best practices when coding." CLAUDE CODE sits between these poles, with a short opening followed by extensive elaboration in later sections.

[5] Three function-calling coders remain in the tree unregistered—one commented out of `coders/__init__.py`, two simply never imported—so `Coder.create()` resolves exactly 13 formats; the April edition inconsistently reported 14.

Verbosity control. Nine of the eleven prompts contain explicit verbosity directives, but the implementations differ sharply. OpenCode's is now the most aggressive prompt in the corpus: "You MUST answer concisely with fewer than 4 lines... One word answers are best," complete with `<example>` blocks ("user: what is 2+2? assistant: 4")—though its Claude-specific prompt, notably, *drops* the quantified rule that the fallback lineage retains. Mistral Vibe keeps its 150-word budget ("Most tasks need under 150 words of prose") and structure-first rule ("Structure first. Prose after, if at all"). Claude Code's response-length rules are quantified ($\leq$ 25 words for inter-tool updates, $\leq$ 100 words for end-of-turn summaries)—though, our re-audit finds, that numeric section is gated to Anthropic-internal builds as an A/B experiment; external users receive qualitative guidance. Codex is qualitative in the 5.2 generation ("concise, direct, and friendly"); the 5.6 prompts fold verbosity into a Personality section, while 5.5 keeps its own length rules. Hermes is qualitative ("Be concise: lead with the change or answer, not a preamble"), Pi minimal (a single bullet: "Be concise in your responses"), Aider requests "a few short sentences" before any patch. Mini-SWE-Agent reaches the same target structurally rather than verbally: exactly one `THOUGHT` block followed by one bash command per turn. OpenHands is alone in leaving verbosity largely to the model: its base prompt says nothing on length, though its GPT-5-family model-specific blocks do instruct concise responses and short preambles.

Banned phrases and forbidden surface forms. Mistral Vibe pioneered the forbidden-phrase list; its rewritten prompt merges two April lists into one: "No filler words: 'robust', 'elegant', 'seamless', 'powerful', 'Great!', 'Absolutely!', 'Of course!', 'Happy to help!' "—a direct attempt to suppress LLM register tics. OpenCode now runs the corpus's second banned-phrase list, model-dependent: its GPT prompt bans openers like "Done —" and "Got it," along with emoji *and em dashes*. Mistral Vibe also retains the strictest emoji prohibition in the corpus: "No emoji of any kind. No smiley faces, icons, flags, or Unicode symbols... This applies to prose, code comments, and commit messages." Claude Code is conditional ("Only use emojis if the user explicitly requests it"), and OpenCode carries the identical conditional rule in its Claude, default, Meta, and Trinity prompts while its GPT prompt bans emoji outright and its GPT-4-era `beast` prompt instead *instructs* emoji status markers; Gemini CLI, Mini-SWE-Agent, Aider, Hermes, and Pi say nothing about emoji. Codex's 5.2 prompt forbids a CLI-specific surface form—inline citations of the shape `[F:README.md L5-L14]` (with bracketed citation glyphs)—because the Codex terminal cannot render them; the rule is absent from the 5.5/5.6 prompts.

Anti-gold-plating directives. A near-universal pattern: scaffolds tell the model not to enlarge the request. Claude Code: "Don't add features, refactor code, or make 'improvements' beyond what was asked," with a separate bullet against speculative abstractions. Mistral Vibe (rephrased since April but preserved in substance): "Change minimally... Don't touch what wasn't asked. Unused imports may have side effects. Redundant-looking code may be load-bearing. When fixing X, leave Y alone." Codex (5.2 generation): "Do not attempt to fix unrelated bugs. ... Fix the problem at the root cause rather than applying surface-level patches"—absent from the 5.4 prompt and dropped entirely by 5.6. OpenHands: "NEVER create multiple versions of the same file with different suffixes (e.g., `file_test.py`, `file_fix.py`, `file_simple.py`)." OpenCode: "NEVER create files unless they're absolutely necessary." Hermes: no drive-by refactors, renames, or reformatting beyond the request. The convergence is striking—six independently developed scaffolds anticipate the same failure mode (LLMs over-deliver on under-specified requests) and respond with phrasing that, while not lifted verbatim, is rhetorically isomorphic. Pi is the deliberate exception: its prompt contains no such directive, policy being delegated wholesale to user-supplied AGENTS.md and extensions.

Интересная синхронность. Видимо, это характерная проблема

Read-before-edit and verify-before-claim. Three prompts make code-reading a precondition for editing. Claude Code: "In general, do not propose changes to code you haven't read." Mistral Vibe strengthened its version between snapshots: "Never edit a file you have not read in this

session. Do not edit a file in the same turn you first read it—read, then act on the next turn." OpenCode's edit-tool description claims enforcement ("This tool will error if you attempt an edit without reading the file")—but no runtime read-tracking exists in the tool's source; the description overclaims, a reminder that prompt text is a behavioral wish, not a mechanism. Claude Code additionally forbids false success claims—"Never claim 'all tests pass' when output shows failures, never suppress or simplify failing checks (tests, lints, type errors) to manufacture a green result"—in a section that, like its word budgets, ships to internal builds first; and Hermes is the only system to *mechanize* the same demand, via the verify-on-stop guard of Section 6 plus a universal anti-fabrication directive ("NEVER substitute plausible-looking fabricated output…"). The other systems treat this as implicit.

Тоже типичный прием против фабрикации "зеленых тестов"

Git commit policy: a convergence that dissolved. In April 2026 this was the corpus's tightest rhetorical convergence: every prompt that mentioned `git` forbade autonomous commits. By July the picture has split three ways. The prohibition survives in Claude Code ("NEVER commit changes unless the user explicitly asks you to"), OpenCode ("NEVER commit changes unless the user explicitly asks"), Hermes ("don't commit, push, or rewrite history unless asked, and never read, print, or commit secrets"), and Codex's 5.2-generation prompt. Mistral Vibe reversed course: it deleted its labeled "Never Commit" hard rule (the changelog reads "Loosened the no-git-commit constraint") and now actively instructs the model how to commit, with a mandated `Co-Authored-By` signature trailer. OpenHands never carried a commit prohibition to reverse—its `<VERSION_CONTROL>` section has taught commit mechanics with a co-author trailer since before the window—while gating *push* and pull-request creation on explicit user request in a separate `<PULL_REQUESTS>` section. And Codex's newest prompts drop the rule entirely: the GPT-5.6 instructions contain no occurrence of "commit." A `codex_git_commit` flag briefly governed commit-attribution behavior but is retired (unused) by the July snapshot—the directive simply left the prompt. What remains universal is the outer boundary: no prompt in the corpus permits autonomous *push*, force-push, or history rewriting. The commit convergence of April, in other words, was not a stable engineering conclusion but a snapshot of trust calibration—and trust moved measurably in one quarter.

Точная фраза

Emphasis markers. The corpus shows three distinct conventions for in-prompt emphasis. Claude Code, Codex, OpenHands, and OpenCode use uppercase tags inline: `IMPORTANT:`, `CRITICAL:`, `NEVER` (OpenCode adds `<system-reminder>` XML injections at runtime). Mini-SWE-Agent uses XML-like tags (`<important>...</important>`), Pi uses XML only as data delimiters, and Hermes combines `MUST`/`NEVER` capitals with checkmark/cross exemplar pairs—shipping its XML-tagged enforcement blocks only to GPT/Codex and Grok models. Aider uses almost none—its enforcement comes from the patch-format examples rather than from emphatic prose (a sparse `IMPORTANT:` survives in the patch-format instructions). Mistral Vibe has moved furthest: no `CRITICAL:` tags and no Hard Rules section remain; emphasis is now fully *structural*, an explicit overridability contract with named "Critical instructions—not overridable" and "Overridable defaults" sections. The contrast suggests three schools: *rhetorical* emphasis (uppercase tags), *structural* emphasis (named rule blocks and precedence contracts), and *example-driven* emphasis (exemplar pairs and format samples), with most systems combining at least two.

интересно, мне казалось, что Claude пишет формула bash скриптов

Tool-use philosophy. Tool guidance varies in specificity. Claude Code insists on dedicated tools over shell: "Do NOT use the Bash tool to run commands when a relevant dedicated tool is provided. … Using dedicated tools allows the user to better understand and review your work." Codex optimizes for parallelism: "Parallelize tool calls whenever possible—especially file reads." Aider's instructions are format-mechanical: "Your entire response containing the patch MUST start with `*** Begin Patch` on a line by itself. … Each file MUST appear only once in the patch." Mini-SWE-Agent is the most restrictive: exactly one bash block per turn, with directory and environment changes prefixed inline (`MY_VAR=val cd /path && ...`) because each call runs in a

new subshell.

What is conspicuously absent. None of the eleven prompts contains explicit harmful-request refusal language. There are no instructions of the form "if the user asks you to do X, refuse and explain why"—the only refusal-shaped text concerns operational risks (destructive git operations, secrets hygiene, dangerous shell patterns), not policy-level harms. The nearest thing in the corpus is refusal *etiquette*: OpenCode's default prompt inherits early Claude Code phrasing instructing the model, *if* it refuses, not to preach about why ("keep your response to 1-2 sentences")—style guidance for a refusal whose grounds are never stated. Safety against misuse is fully delegated to the underlying model's pretraining and any provider-side policy layer. This is a notable design choice: production scaffolds, even those bundled with their own provider, do not duplicate alignment work in the system prompt—and the finding now holds at 11/11 across three more vendors and community lineages.

Observation 3. Prompt rhetoric converges where engineering experience converges, then *thins* as trust calibrates. Anti-gold-plating language appears in nearly isomorphic phrasings across six independently developed scaffolds, and the no-autonomous-commit rule was universal in April 2026. By July, the corpus shows the rules' life cycle: Mistral Vibe reversed its commit prohibition into commit *instruction* with a mandated co-author trailer (OpenHands had taught commit mechanics all along), and Codex's newest model generation dropped both the commit rule and the anti-gold-plating directives from its prompts—behavioral policy thinning as models internalize the norms. Three rhetorical strategies now coexist: harness-enforced policy prose (Claude Code, OpenCode, Hermes), structural contracts (Mistral Vibe's seven-level instruction hierarchy and overridability sections), and near-total delegation to user context (Pi's 30-line prompt). Model-conditional rhetoric has emerged as its own axis: Hermes gates enforcement blocks by model family with comments citing observed per-model failures, and OpenCode maintains nine per-family prompts whose strictness varies by recipient. The stable invariant across all eleven: no policy-level refusal language anywhere—alignment work is never duplicated in the harness prompt.

7.4 Streaming and Response Processing

Response processing strategies span a wide range:

- **Claude Code**: SSE with fine-grained events (`content_block_start/delta/stop`, `message_delta`). Supports extended thinking via `thinking` blocks with configurable `budget_tokens`.
- **Codex**: WebSocket + SSE fallback. `ResponseItem` events parsed from JSON-RPC. Connection prewarm with `generate=false`. Turn state maintained via `x-codex-turn-state` header.
- **Gemini CLI**: `@google/genai` SDK `chat.sendMessageStream()` with a four-attempt mid-stream retry loop (`MidStreamRetryOptions`) recovering from content, network, and invalid-stream errors. Wraps the SDK in a custom `GeminiChat` class to work around a function-response validation bug. Keeps extended-thinking parts out of persisted history at response-recording time so they don't pollute future cache lookups.
- **Mistral Vibe**: Mistral SDK `stream_async()` aggregates `LLMChunk` blocks; `ThinkChunk` blocks are routed into a separate `reasoning_content` field. Aggregated usage stats are produced across the stream.
- **Aider**: Token-level streaming with `mdstream` for rich terminal rendering. Retry loop with exponential backoff; handles `FinishReasonLength` by resuming with assistant prefill.
- **Mini-SWE-Agent**: Full response (no streaming). Simplest approach; relies on model's native completion.

- **OpenHands**: SSE via LiteLLM wrapper, now with first-class streaming (an `LLM.stream` field and token deltas forwarded through `on_token` callbacks) plus an OpenAI Responses path. Supports prompt caching markers for Anthropic models.
- **Hermes**: five owned transport implementations; a ∼23-value `FailoverReason` enum drives classified recovery (compress vs. rotate credential vs. fallback chain vs. abort).
- **Pi**: nine wire-protocol implementations with a `partial-json` salvage parser for streamed tool arguments—paired with the truncation guard of Section 6, since salvaged arguments can validate while incomplete.
- **OpenCode**: Vercel AI SDK `streamText` owns dispatch; token-granular part deltas are persisted and re-served over SSE by the embedded server, so every client (TUI, web, IDE) replays the same stream.
- **OpenClaw**: ACP delta events for text, thought, and tool calls; provider-specific transport streams handle the underlying SSE/WebSocket layer.

7.5 Advanced API Features

Интересная формулировка

The single-provider systems exploit features unavailable to multi-provider abstractions:

Claude Code's extended-thinking integration exposes a `budget_tokens` parameter that controls how much internal reasoning the model performs before responding; while LiteLLM does pass through Anthropic's `thinking` block, surfacing `budget_tokens` as a first-class scaffold knob still requires Anthropic-specific conditional code in any multi-provider design. Codex's reasoning-effort enum spans `minimal/low/medium/high/xhigh`, extended in mid-2026 with `max` and `ultra`—and `ultra` is not just more reasoning: the model catalog describes it as maximum reasoning *with automatic sub-agent task delegation*, an effort level that changes orchestration behavior, unique in the corpus. Mistral Vibe's per-model `thinking` field now spans five levels (`off/low/medium/high/max`) mapped onto Mistral's `reasoning_effort` enum, and Pi normalizes the whole landscape behind a unified seven-level scale translated through ten per-API `thinkingFormat` dialects. Codex implements prompt caching via a `prompt_cache_key` derived from the session's `ThreadId` (with an explicit override hook for delegate and guardian sessions); this is a lighter-weight strategy than Claude Code's Blake2b-hashed static/dynamic boundary. Aider's vision support warrants a note: when the active model's `supports_vision` attribute is set (via LiteLLM metadata), Aider accepts image files in the chat, handles MIME types, and passes them to the model. The support is model-gated rather than unconditional, hence marked "Opt." in Table 17; Mistral Vibe joined the vision column in v2.14–2.18 (TUI @-mentions, clipboard paste, ACP inline image blocks), and Mini-SWE-Agent retains an opt-in, default-off multimodal path (a `multimodal_regex` expanding tagged content into image blocks), so no corpus system is unconditionally text-only. Gemini CLI exposes extended thinking (with thought-part stripping to keep caches clean) and then routes the choice of model itself through `ModelRouterService`; that treatment of per-request model selection as classifier-driven runtime scheduling remains unique in the corpus (OpenHands's RouterLLM profiles route by configuration, not per-query classification), now extending to a managed *on-device* runtime (a `gemini gemma` command group provisions a local LiteRT-LM server, and the routing chain can use the local Gemma model as its query classifier). On the observability side, Gemini CLI emits OpenTelemetry traces alongside per-token cost accounting. Mistral Vibe also instruments its agent loop, tool execution, and hooks with OpenTelemetry (`agent_span/tool_span/hook_span` async context managers with an OTLP exporter and GenAI semantic-convention attributes), and Pi contributes the corpus's most unusual observability metric: a per-turn *cache-miss dollar-waste audit* that prices every avoidable cache invalidation.

Вот как в настройки reasoning пролезла настройки мультимодальности

8 Tool and Action Systems

The tool system defines what the agent can actually do: which actions it can invoke and how they are executed. This subsystem absorbs a large share of the engineering effort in every mature agent we studied.

8.1 Tool Count Spectrum

Figure 3 illustrates the spectrum from minimalist to maximal tool designs. The systems span from a single bash tool to 109+ tools including media processing and channel integrations—with Pi making the sparse end a philosophy (seven tools built, four exposed by default, everything else an extension) and Hermes pushing the general-purpose end to 69 built-ins behind a toolset layer that scopes what each platform's model actually sees.

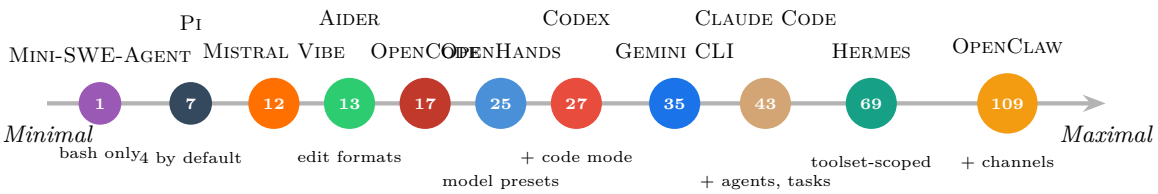


Figure 3: Tool count spectrum (July 2026). Numbers indicate distinct built-in tools or edit formats; counts for OpenHands and Codex are default surfaces (their full catalogs reach 30+). Pi builds 7 and exposes 4 by default; Hermes registers 69 but scopes visibility per platform through a toolset layer.

8.2 Tool Interface Architectures

Tool interfaces range from almost no abstraction—Mini-SWE-Agent sends everything through a single shell call (`subprocess.Popen(shell=True)`, wrapped only to kill the process group on timeout)—to rich typed contracts. Claude Code's 43 tools each declare validation, permission checking, concurrency safety, and UI rendering as separate interface concerns. Codex uses Rust trait objects for runtime polymorphism—the registry stores `Arc<dyn CoreToolRuntime>` where `CoreToolRuntime: ToolExecutor`, the abstraction now extracted into a standalone `codex-tools` crate shared with code mode and extensions. Gemini CLI routes calls through a state-machine `Scheduler` that walks each invocation through deterministic *Validating* → *Executing* → *Completed/Errored* phases. Mistral Vibe adds *tree-sitter-parsed bash*—an `asyncio.create_subprocess_shell`-based execution path that uses `tree_sitter_bash` to validate command structure before spawning—and fine-grained permission scopes (command pattern, file pattern, URL allowlist, directory containment). Hermes's singleton registry separates *registration* from *exposure*: 69 tools self-register at import, a toolset layer scopes what each platform's model sees, per-tool availability probes (30-second TTL with a last-good grace window) keep one flaky `docker version` from stripping a toolset, and a coercion layer repairs schema drift (`"42"`→42) motivated by observed open-model output. Pi gives each of its seven tools a pluggable `Operations` interface (`BashOperations`, `EditOperations`, ...)—the single remoting seam through which SSH, container, and micro-VM extensions relocate execution without touching the tools. OpenCode registers 17 first-party tools with Effect schemas and swaps the *tool surface* per model: GPT-family models receive a port of Codex's `apply_patch` DSL and lose `edit/write` entirely (Section 8.4). The spectrum reflects the simplicity-versus-capability axis: more interface surface means more safety guarantees but more engineering cost.

8.3 Deferred Tool Loading

Claude Code introduced *deferred tool loading*: tools marked with `shouldDefer = true` are excluded from the initial system prompt. The LLM discovers them on-demand via `ToolSearchTool`,

which supports keyword search and direct selection (`select:<tool_name>`). This reduces prompt size significantly: of 43 tools, only a core subset is loaded initially. The deferred tool set is cached via `getDeferredToolsCacheKey()`, invalidated when the set changes.

The April edition presented this as a Claude Code exclusive; by July it is a spreading pattern with independent implementations. Codex marks tools (by default, MCP tools) with a `defer_loading` flag and exposes a `tool_search` tool backed by a *BM25 lexical index* over tool specs—the same prompt economics with ranked search instead of substring matching. Hermes generalizes the idea with a threshold: when MCP and plugin schemas would exceed 10% of the context window, they collapse into three bridge tools (`tool_search`/`tool_describe`/`tool_call`) searched by an inlined BM25 engine. OpenCode defers *skills* (catalog lists name and description only; a native `skill` tool fetches bodies) and treats its experimental code mode as MCP-catalog deferral: instead of exposing every MCP tool, one `execute` tool runs model-written scripts against the catalog. Mistral Vibe's `defer_mcp` flag, by contrast, defers only MCP *connection* at startup for latency, not prompt visibility—once integrated, every MCP tool is model-visible, so we no longer group it with the prompt-level deferral pattern.

8.4 File Editing Strategies

File editing is the core action for SWE agents. The eleven systems implement eight fundamentally different strategies, summarized in Table 7.

Table 7: File editing strategies across the eleven systems (July 2026).

System	Format	Matching	Fallback	Innovation
Claude Code	Exact string replacement	Unique substring in file	Error + context	Zod schema validation
Codex	`*** Begin/End Patch`	Unified diff hunks	Hunk-level retry	Custom patch DSL; Lark-grammar-constrained variant
Aider	13 formats (SEARCH/REPLACE, udiff, diff-fenced, . . .)	Exact → fuzzy (dmp)	Similar line suggestions	RelativeIndenter
OpenHands	str_replace_editor (5 cmds) + `apply_patch` + Gemini-ported `edit`	Exact unique match	Error message	undo_edit; three coexisting edit dialects
Gemini CLI	`edit` (old/new string) + `write_file`	Cascading: exact → flexible → regex → fuzzy	LLM "edit fixer" subcall repairs the pair	LLM-assisted edit repair
Mistral Vibe	Exact `edit` (old/new string, `replace_all`) + create-only `write_file`	Exact unique substring	Tool error on ambiguity	Migrated *off* fuzzy SEARCH/REPLACE (Apr→Jul)
Mini-SWE-Agent	sed/awk via bash	Regex-based	Shell error	None (model-driven)
Hermes	One `patch` tool: SEARCH/REPLACE + V4A DSL (+ `write_file`)	9-strategy fuzzy chain (exact → . . . → block-anchor 0.50/0.70 → context-aware 0.80)	Closest-line suggestions; escalate to `write_file` after 3 failures	Two-phase validate-then-apply; backend-transparent shell file ops
Pi	Multi-edit exact replacement (`edits[]`, overlap rejection)	Exact → Unicode/whitespace canonicalization (no similarity threshold)	Uniqueness guidance error	Byte-fidelity overlay; async diff preview before execution
OpenCode	Model-conditional: `apply_patch` DSL (GPT family) or SEARCH/REPLACE `edit` (others)	9-stage replacer cascade (Levenshtein 0.65) / 4-pass patch relaxation	Distinct not-found vs. ambiguous errors	Editing dialect swapped per model in the registry; LSP diagnostics appended to results
OpenClaw	N/A	N/A	N/A	Not a code editor

На часть быстро меняется

The editing landscape reorganized between snapshots. The April edition described Mistral Vibe and Gemini CLI as a "polymorphic-by-degree" pair—coarse whole-file tool plus fuzzy fine-grained patch tool. Neither half survived the quarter. Mistral Vibe deleted its SEARCH/REPLACE tool outright and converged on Claude Code's exact unique-substring contract (with a `replace_all` escape hatch and a now create-only `write_file`): a rare observable case of a production harness migrating between editing-strategy clusters, and evidence that stronger models shift the optimum from tool-side drift tolerance toward strict contracts.[6] Gemini CLI's `edit`, on closer inspection, is an old/new-string tool with a cascading matcher (exact → whitespace-flexible → regex → fuzzy) that ends in something no other system had: an *LLM edit-fixer subcall* that repairs the failed old/new pair against a model-supplied `instruction`—the corpus's first in-tool LLM-assisted edit repair.

The fuzzy-cascade family and its visible lineage. Where Mistral Vibe left, two newcomers arrived with the corpus's most elaborate drift tolerance, and their source code documents its ancestry. OpenCode's nine-stage replacer cascade (exact → line-trimmed → block-anchor → whitespace/indentation/escape-normalized → context-aware, under Levenshtein 0.65 with a disproportionate-match guard) carries header comments crediting Cline and Gemini CLI evals; Hermes's nine-strategy chain is annotated "inspired by OpenCode." Editing machinery now has a visible cross-harness genealogy—convergence by inheritance, not just rediscovery. Pi stakes out a third position: no similarity threshold at all, only Unicode/whitespace *canonicalization* (NFKC, smart quotes, trailing whitespace), with replacements computed in normalized space overlaid linewise onto the original bytes so untouched lines keep their exact original whitespace—plus an async diff preview rendered before the tool executes.

Aider's `RelativeIndenter` still deserves special attention. It converts absolute indentation to relative changes using Unicode markers (default: left arrow ←, selected to avoid collisions with file content). The `make_relative()` method tracks indent differences between consecutive lines, and `make_absolute()` reconstructs original indentation. This enables edit blocks to work across different indentation levels, a common failure mode for other systems. Combined with fuzzy matching via `diff_match_patch` (threshold 0.95, distance 500), this creates a robust editing pipeline.

Observation 4. The file-editing strategy is one of the most important determinants of code-modification accuracy in an SWE agent, and model-aware polymorphism is no longer Aider's alone. Aider selects the edit *format* per model through a prompt-class factory; OpenCode reaches the same insight through the tool registry, swapping the *toolset* per model (GPT-family models get a Codex-style patch DSL and lose the string-replacement tools entirely). The corpus's editing machinery now shows explicit cross-harness lineage (Hermes's matcher "inspired by OpenCode"; OpenCode's cascade crediting Cline and Gemini CLI), and it is in visible motion: Mistral Vibe migrated from fuzzy SEARCH/REPLACE to Claude Code-style exact matching within one quarter, while Gemini CLI added LLM-assisted edit repair. Aider's `RelativeIndenter` remains the most sophisticated solution to indentation-sensitive matching.

8.5 Execution and Sandboxing

Table 8 compares sandboxing approaches in detail.

[6] The April text also overstated the old mechanism: re-audit shows v2.7.5's ≥0.90 fuzzy matcher only generated "closest match" diagnostics for error messages; it never applied fuzzy edits.

Table 8: Execution sandboxing mechanisms (July 2026).

System	Mechanism	Isolation Level	Investment
OpenHands	`Workspace` abstraction: Docker, Apptainer, remote-API, cloud workspaces (KVM passthrough app-side)	Process + filesystem + network	Substantial
Codex	Bubblewrap (vendored, compiled in-tree) + Landlock legacy fallback (Linux), Seatbelt (macOS), restricted tokens (Windows)	OS-level namespaces + filesystem + network	Substantial
Gemini CLI	Docker / Bubblewrap / gVisor / LXC (Linux), Seatbelt via `sandbox-exec` (macOS), restricted tokens (Windows); TOML policies	OS-level + per-mode policy + env scrubbing	Moderate
Claude Code	Opt-in OS sandbox via Anthropic's `sandbox-runtime` (Bubblewrap/Seatbelt) with network restrictions; git worktrees for branch isolation	OS-level (opt-in) + filesystem	Moderate
Mini-SWE-Agent	Docker, Singularity, Bubblewrap (pluggable)	Configurable	Light
Hermes	Six pluggable execution backends (local, Docker, SSH, Singularity, Modal, Daytona); local backend = policy floor, no OS primitives	Per-backend (container/VM); none locally	Delegated
Mistral Vibe	None OS-level; opt-in `-worktree` branch isolation	Filesystem (branch-level, opt-in)	None
Pi	None built-in, with documented rationale; opt-in extensions (Anthropic `sandbox-runtime`, QEMU micro-VM) via per-tool `Operations` seam	None by default	None (extension-space)
OpenCode	None (permission rules + tree-sitter command parsing; shadow-git checkpoints; "sandbox" in-tree means worktrees)	None	None (policy-only)
Aider	None (direct local execution)	None	None
OpenClaw	None (local gateway execution)	None	None

Codex is the most serious sandboxing investment in the corpus, now packaged as a dedicated `sandboxing` crate. On Linux it runs Bubblewrap—since mid-2026 *vendored and compiled in-tree*, invoked through C FFI—for namespace isolation (`-ro-bind`, `-unshare-net`, `-unshare-user`, `-unshare-pid`), with Landlock demoted to a legacy fallback behind a feature flag. On macOS

it drives Seatbelt profiles through `/usr/bin/sandbox-exec`. On Windows it uses restricted-token processes. Protected paths (`.git`, `.agents`, `.codex`) are re-marked read-only even when they sit inside a writable root. The result is OS-level isolation without the overhead of a container runtime.

Gemini CLI joins Codex as the second cross-platform sandboxer. Gemini CLI's `SandboxManager` implements the same three-platform stack as Codex: Docker or Bubblewrap on Linux, Seatbelt via `/usr/bin/sandbox-exec` on macOS, restricted-token processes on Windows. Where Codex bakes its policy into Starlark `execpolicy` rules (Section 10), Gemini CLI layers a TOML-based per-mode sandbox policy (`plan` / `default` / `accepting_edits`) plus environment scrubbing of provider credentials (name-pattern regexes over `TOKEN/SECRET/KEY/AUTH/CREDENTIAL`, a denylist, and GitHub-token value regexes) before each command runs. Gemini CLI's sandbox investment is meaningfully lighter than Codex's because Gemini CLI reuses Node's child-process API and OS-provided binaries rather than implementing low-level namespace plumbing itself; Codex pays for its larger footprint in vendored-Bubblewrap integration and Rust-native namespace control. The April edition observed that the two largest systems in the corpus were also the two with native cross-platform sandboxing and asked whether the correlation was structural; the expanded corpus answers *no*: Hermes (∼642K lines) ships zero OS-level isolation primitives, delegating containment to six pluggable execution backends while investing its safety budget in content-borne threats, and OpenCode (∼578K lines) is policy-only. Size predicts *some* large safety investment, not specifically a sandbox; Section 10 maps where each system spends instead.

9 Memory and Context Management

Every agent we looked at runs into the same problem eventually: the conversation outgrows the context window. The literature offers two broad families of solutions: tiered memory architectures (such as MemGPT [74], which shuffles data between an in-context working set and an external store) and prompt-compression methods (such as LLMLingua [75], which shortens text before it reaches the model). The eleven systems in the corpus implement four practical strategies (Figure 4), which we present in roughly increasing order of sophistication.

9.1 Strategy Taxonomy

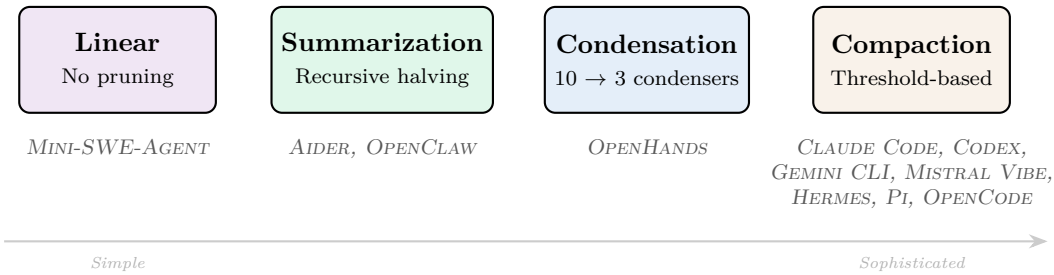


Figure 4: Memory management strategies ordered by sophistication. Seven of the eleven systems—the four provider-native harnesses and all three systems added in this revision—converge on threshold-triggered LLM compaction: a de facto standard for production-grade context management, though each newcomer hybridizes it with traits of the other strategies (iterative summary merging, pluggable engines, session-tree substrates).

Mini-SWE-Agent как муха дрозофила в мире harness

9.2 Linear History (Mini-SWE-Agent)

Mini-SWE-Agent makes no attempt to manage context. The full message list grows unboundedly, relying on the LLM's native context window (up to 1M tokens with the Claude 4 family). This simplicity enables perfect reproducibility and straightforward trajectory analysis, a deliberate choice for a research baseline system.

9.3 Recursive Summarization (Aider)

Aider's `ChatSummary` class implements a recursive halving algorithm. When the message history exceeds `max_tokens` (default: 1024), it splits messages into two halves by token count, keeping the tail (most recent 50%) intact. The head is summarized via an LLM call (`summarize_all()`). If the combined summary + tail still exceeds the limit, the process recurses up to 3 levels. A separate "weak model" (e.g., GPT-3.5-turbo) can configured for cost-efficient summarization.

9.4 Pluggable Condensation (OpenHands)

OpenHands pioneered pluggable memory management through an abstract `Condenser` base class. The V0 application shipped ten implementations; the V1 SDK consolidated the zoo to three (`NoOpCondenser`, `LLMSummarizingCondenser`, `PipelineCondenser`) over a `RollingCondenser` base—a telling simplification, most of the specialized variants having proven unnecessary. Two upgrades came with the consolidation: condensation is itself *event-sourced* (`CondensationRequest`/`Condensation` events in the log, so compression history is replayable), and the condenser doubles as *error recovery*—context-window overflows and malformed-history errors trigger a condensation rather than a crash. The summarizing condenser also gained a token-count trigger (`max_tokens`) alongside its event-count threshold, aligning OpenHands with the threshold-compaction family below.

9.5 Threshold Compaction (Seven Systems)

Самый зрелый к популярный подход!

Claude Code's compaction triggers when the running token estimate enters a buffer below the model's effective context window (`AUTOCOMPACT_BUFFER_TOKENS = 13 000`, with an env-overridable percent threshold for testing). The process: (1) strip images from messages (replace with `[image]` markers), (2) group messages by API round, (3) invoke the LLM to generate a conversation summary, (4) create a `SystemCompactBoundaryMessage` separating pre-compaction history from the summary. Post-compaction cleanup restores up to 5 files (`POST_COMPACT_TOKEN_BUDGET = 50 000`) and truncates skills to a per-skill 25 000-token budget.

Codex supports both local in-process and cloud-based summarization via a dedicated `/responses/compact` endpoint, with `model_auto_compact_token_limit` controlling when summarization triggers (the separate `/memories/trace_summarize` endpoint serves the memories pipeline of Section 9.6); a remote-compaction v2 path and per-thread rollout token budgets (with remaining-budget reminders and turn aborts on exhaustion) were added in the spring 2026 releases.

Gemini CLI: Multi-Stage Context Pipeline. Gemini CLI's context management is more sophisticated than simple compression. At the top level, a `ChatCompressionService` fires when history exceeds 50% of the model's token limit, calls Gemini to generate a state snapshot, and *preserves the most recent 30%* of history verbatim (a token-budget truncation pre-pass and a `PreCompress` hook now fire before compression). Beneath this, a dedicated context-processors module implements a multi-stage context distillation pipeline with pluggable processors (rolling summary, node distillation, node truncation, blob degradation, tool masking, state snapshot), consolidated in v0.43–0.45 into a `ContextManager` that maintains a *pristine-versus-active* context graph in a working buffer, runs processors through an orchestrator with hysteresis-based trigger damping, and replaces static chars-per-token estimates with an adaptive token calculator calibrated against observed API token counts. A separate `ContextCompressionService` tracks per-file compression levels (FULL, PARTIAL, SUMMARY, EXCLUDED), providing finer granularity than Claude Code's binary compacted/uncompacted split, and a pluggable `AgentHistoryProvider` interface lets extensions substitute their own compression logic.

Mistral Vibe: Middleware-Triggered, Now Two-Tier and Reactive. Mistral Vibe's `AutoCompactMiddleware` checks `context_tokens >= auto_compact_threshold` before each turn; the compaction routine itself, extracted into a `CompactionManager`, has grown three refinements since April: the post-compaction state is a context *envelope* that re-injects prior user messages

alongside the summary (so original task goals survive resets), a cache-friendly primary summarization falls back to a dedicated tool-less summarizer, and the same routine fires *reactively* when a turn overflows the context window mid-flight. The integration through the middleware pipeline remains architecturally cleaner than in-loop checks: compaction is just another turn policy. MISTRAL VIBE's complementary `RewindManager` creates filesystem-level checkpoints per user message and supports rewinding with optional file restoration—now forkable to a new session and exposed *over ACP* so IDE clients can drive the rewind. Once unique, checkpointing now has a finer-grained sibling: OPENCODE maintains a *shadow git repository* per project (separate `-git-dir` over the real worktree) snapshotting at pre-stream, step start, and step finish, yielding per-step patch parts and full conversation-plus-filesystem revert.

Hermes: Lineage Compaction. HERMES lands in the threshold family (trigger at 50% of context minus max-tokens, 64K floor; protect the first three messages plus a token-budgeted tail; iterative auxiliary-model summary capped at min(5% of context, 12K tokens)) but adds a construct no other system has: compaction does not rewrite the transcript—it *ends the session*. Each compaction closes the current SQLite session (`end_reason='compression'`) and rotates to a child chained by `parent_session_id`; lineage helpers walk the ancestry, and session search deduplicates across it. Context management and session persistence become one mechanism, and no history is ever destroyed. The engine is pluggable (a `ContextEngine` ABC), echoing OPENHANDS's condensers.

Pi: Compaction over a Session Tree. PI triggers at `contextWindow` − 16,384 tokens (keeping 20K recent), with two traits borrowed from the other strategies: summaries are *iteratively merged*—the update prompt re-ingests the previous summary rather than re-summarizing from scratch (a recursive-summarization trait)—and a `session_before_compact` hook lets extensions veto or replace the result entirely (a pluggable-condensation trait). The distinctive substrate is the session itself: an append-only JSONL *tree* where every entry carries `id`/`parentId` and a movable leaf pointer defines the active branch. `/tree` navigation, `/fork`, and `/clone` unify what other harnesses implement as three separate features (checkpointing, rewind, alternative exploration), and abandoning a branch optionally generates an LLM *branch summary* spliced into the new position, so exploration is never lost—with the explicit trade-off that filesystem state is not restored. Cumulative `<read-files>`/`<modified-files>` lists persist as structured memory across compactions.

OpenCode: Anchored Incremental Summaries. OPENCODE triggers at `limit.input` minus a reserved-output margin, using a chars/4 estimate (no tokenizer exists anywhere in the tree), and produces the corpus's most explicitly *incremental* summary: the previous summary is passed back in a `<previous-summary>` block and merged ("preserve still-true details, remove stale details") by a hidden, all-tools-denied `compaction` agent, into mandated Markdown sections (Objective / Important Details / Work State / Next Move / Relevant Files). A verbatim tail of two user turns is preserved; opt-in pruning soft-deletes old tool outputs, protecting the newest 40K tokens and firing only if more than 20K are reclaimable.

9.6 Persistent Memory Pipelines

The April edition of this study could describe cross-session memory in one sentence per system: a Markdown context file here, a transcript store there. Three months later, persistent memory is where the provider-native systems are differentiating most aggressively, and the designs diverge instructively on a single axis: *who writes the memory, and who reviews it.*

CODEX runs the most autonomous design. A background two-phase pipeline extracts structured memories from recent session rollouts (per-rollout LLM extraction into a SQLite state database, with leased jobs, retry backoff, and secret redaction) and then *consolidates* them into filesystem artifacts under a git-baselined `~/.codex/memories/` root—the consolidation performed by a sandboxed internal sub-agent (no approvals, no network, local-write-only) reviewing a git-style workspace diff.

Memories are injected into new sessions with citation tracking and usage-based ranking. CODEX is thus the first system in the corpus whose long-term memory is maintained *by an agent* rather than by code; an experimental "chronicle" sidecar extends the same machinery toward passive screen context.

GEMINI CLI moved in the opposite governance direction. Its April-era callable `save_memory` tool is gone—the system prompt now states flatly that no such tool exists—replaced by (a) the model editing `GEMINI.md` or a private per-project `MEMORY.md` index directly with ordinary edit tools, and (b) an asynchronous *skill-extraction sub-agent* that mines completed session transcripts and emits canonical patch files into a per-project `.inbox/`, which the user reviews and applies via `/memory`. Where CODEX lets an agent consolidate memory autonomously, GEMINI CLI inserts a human-gated review inbox between extraction and persistence—the only such design in the corpus.

HERMES keeps persistent memory deliberately tiny and cache-friendly: two bounded Markdown files (`MEMORY.md`, 2,200 characters; `USER.md`, 1,375) injected as a *frozen snapshot*, so mid-session writes hit disk without invalidating the prompt cache. Conversation recall is deterministic—a `session_search` tool over trigger-maintained SQLite FTS5 tables (BM25, plus trigram indexing for CJK), "no LLM calls anywhere"—and embeddings exist only in opt-in memory plugins. OPENCLAW ships an *Active Memory* plugin (present, we note for the record, since the very April release our earlier edition audited) that runs a dedicated memory sub-agent immediately *before* the main reply, plus its older opt-in LanceDB conversation-memory extension. OPENHANDS folds memory into its context-file convention: a `<MEMORY>` prompt section instructs the model to persist durable facts into `AGENTS.md` itself. CLAUDE CODE persists per-project memory directories of Markdown facts; PI and MINI-SWE-AGENT deliberately persist nothing beyond the session substrate.

Observation 5. Persistent memory has replaced compaction as the frontier of context engineering. Compaction designs have converged (seven of eleven systems use threshold-triggered LLM summarization, increasingly with incremental merging); what now differentiates the field is the memory write path. Four governance models coexist: agent-maintained (CODEX: a sandboxed sub-agent consolidates memories over a git-baselined store), human-gated (GEMINI CLI: extraction sub-agent writes to a patch inbox the user reviews via `/memory`), model-direct-but-bounded (HERMES: character-capped Markdown snapshots, frozen per session for cache hygiene; OPENHANDS, CLAUDE CODE: context-file conventions), and pre-turn agentic recall (OPENCLAW's Active Memory sub-agent). Notably, none of the eleven uses embedding-based retrieval as its primary memory substrate—the deterministic-retrieval finding of Observation 9 extends from code to memory, with SQLite full-text search (HERMES) as the production ceiling.

9.7 Repository Context

Beyond conversation history, agents must manage code context:

- **Aider RepoMap**: Tree-sitter-based symbol indexing with diskcache (SQLite v4)—still the only ranked repository map in the corpus. Symbols are ranked by relevance to the current conversation. Token budget is adaptive (`map_tokens`, default 1024) with a `map_mul_no_files` multiplier for large repositories.
- **Claude Code Memory System**: Auto-discovers `CLAUDE.md` files via a dedicated `claudemd` subsystem (managed, user, and project scopes traversed from the working directory), injected as nested memory attachments with deduplication; the separate `memdir` machinery manages the per-project persistent auto-memory directory.
- **Codex Instructions**: Hierarchical `AGENTS.md` instructions—precisely, a *concatenation* from the project root down to the working directory (nested content appended last, so "precedence" holds only by position), plus an `AGENTS.override.md` escape hatch and configurable root markers and fallback filenames.
- **Gemini CLI Context Files**: Three-scope `GEMINI.md` merge (global, extension, project)

gated by folder trust, plus JIT injection of subdirectory context files through tool output and the private `MEMORY.md` project index.

- **Mistral Vibe JIT Discovery**: Top-level `AGENTS.md` in the prompt (user and project scopes), and—since v2.19—nested `AGENTS.md` files surfaced just-in-time when `read_file` touches files beneath them.
- **Hermes Hierarchy with Threat Scanning**: First-found-wins walk (`.hermes.md/HERMES.md` to the git root, then `AGENTS.md`, `CLAUDE.md`, `.cursorrules`); every context file is scanned for prompt-injection patterns and *blocked wholesale* on a match; a subdirectory hint tracker appends per-directory context files to tool results, keeping the cached prompt untouched.
- **Pi Ancestor Walk**: `AGENTS.md/CLAUDE.md` collected from cwd to root into `<project_context>` tags; loading of repo-controlled *configuration* (extensions, skills) is trust-gated, but context files load regardless—a documented, deliberate injection-surface acceptance.
- **OpenCode Pull-Based Attach**: `AGENTS.md/CLAUDE.md/CONTEXT.md` walk-up (first filename wins) plus remote-URL instruction files; nested `AGENTS.md` lazily attach when the `read` tool touches files beneath them.
- **OpenHands Third-Party Ingestion**: the skills subsystem ingests `AGENTS.md`, `.cursorrules`, and nested per-directory `AGENTS.md` as scoped rules—one system reading three other ecosystems' conventions.
- **OpenClaw Transcripts**: JSONL transcript files persisted per agent/session at `~/.openclaw/agents/<agentId>/sessions/<sessionKey>.jsonl`.

10 Safety and Permission Models

Safety mechanisms become important in direct proportion to the autonomy the agent is given. A recent line of academic work has formalized the threat model: indirect prompt injection through file or web content [77], AgentDojo [76] as a dynamic evaluation environment for injection attacks and defenses, risk-awareness benchmarks like R-Judge [78], and ToolSword's [79] catalog of tool-use safety failure modes. The corpus covers a wide range, from systems with essentially no safety controls to multi-layered architectures.

10.1 Claude Code: Three-Layered Permission System

Claude Code implements the most architecturally elegant safety system, with three layers (Figure 5):

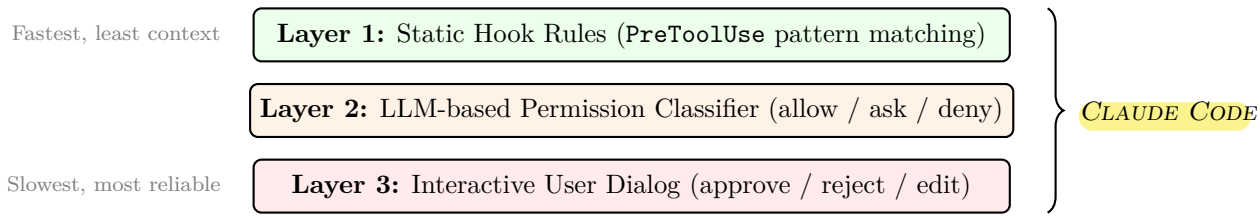


Figure 5: Claude Code's three-layered permission system.

For *background* and forked sub-agents, only layers 1–2 are available—their would-be prompts resolve to denials—while default foreground sub-agents can still surface interactive dialogs. Unresolvable cases escalate to the parent coordinator. A **denial tracking** mechanism counts permission refusals per session; after N denials, the system falls back to prompting. Each async agent maintains local denial state to prevent cross-agent pollution.

10.2 Codex: A Four-Layer Permission Stack with Native Sandboxing

The April edition described CODEX's safety as two mechanisms—policy rules plus OS sandbox—and characterized its approval gate as "near-invisible." The July snapshot shows four layers, two of them converging on CLAUDE CODE's design:

Layer 1: Execution Policy (Starlark, not TOML). Executable rules (`execpolicy/`) are written in *Starlark*: `prefix_rule(pattern=[...], decision="allow|prompt|forbidden", match=[...], not_match=[...])` and `network_rule(...)` functions, loaded from `CODEX_HOME/rules/` and per-config-layer rules directories.[7] A distinctive detail: rules may carry inline `match`/`not_match` examples that are *validated at parse time*—executable test cases inside the policy file.

```
prefix_rule(
    pattern = ["git", "status"],
    decision = "allow",
    match = [["git", "status"], ["git", "status", "--short"]],
    not_match = [["git", "push"]],
)
network_rule(
    protocol = "https",
    host = "*.github.com",
    decision = "allow",
)
```

Listing 2: Codex execution policy example (Starlark).

Layer 2: Lifecycle Hooks. A hooks crate exposes external hooks whose event vocabulary is near-verbatim CLAUDE CODE's: `PreToolUse`, `PermissionRequest`, `PostToolUse`, `PreCompact`, `SessionStart`, `UserPromptSubmit`, `SubagentStart`, `SubagentStop`, `Stop`—plus a `PostCompact` event with no CLAUDE CODE counterpart—with block/allow decision outputs. Section 14.5 discusses this as the corpus's clearest case of one provider-native harness adopting another's extension interface as a de facto standard.

Layer 3: Guardian, an LLM Approval Reviewer. When a command would require user approval, a dedicated *Guardian* session re-assesses the exact planned action against a policy prompt (compact transcript reconstruction, strict-JSON verdict), fails *closed* on timeout or malformed output, and carries a per-turn rejection circuit breaker. The "no LLM classifier" contrast the April edition drew against CLAUDE CODE no longer holds: both provider-native flagships now implement the static-rules → LLM-classifier → interactive/OS backstop pattern. A user-facing collaboration-mode selector (Plan/Default presets that mask model, reasoning effort, and developer instructions) completes the convergence: all four provider-native systems now ship a read-only planning mode.

Layer 4: Native OS Sandboxing. The sandbox implementation spans three platforms:

- **Linux**: vendored, in-tree Bubblewrap (invoked via C FFI) with read-only filesystem (`-ro-bind / /`), writable roots via `-bind`, network namespace isolation (`-unshare-net`), user/PID namespace isolation; Landlock demoted to a legacy fallback.
- **macOS**: Seatbelt profiles via `/usr/bin/sandbox-exec` with configurable allow/deny rules.
- **Windows**: Restricted-token processes with ACL-based read/write restrictions.

[7] The April edition rendered these rules as TOML; that was an error in our earlier audit—the policy language has been Starlark throughout.

Observation 6. OS-level sandboxing is one of the most code-expensive capabilities in the corpus, and it is a *choice* rather than a consequence of scale. The April corpus suggested a size correlation (its two largest systems were its two cross-platform sandboxers); the expanded corpus breaks it. HERMES, among the largest systems here, ships zero OS-level isolation primitives: it delegates containment to six pluggable execution backends and spends its safety budget on content-borne threats instead (promptware scanning—pattern-matching for prompt-injection payloads—a skills supply chain, a command-policy floor that survives `-yolo`). OPENCODE, comparably large, is policy-only, investing in syntax-aware command permissioning rather than isolation. PI documents the refusal as a security argument: a partial in-process sandbox "would be easy to misunderstand as a security boundary." What remains true is the cost side: where native sandboxing is built (CODEX, GEMINI CLI; CLAUDE CODE wraps Anthropic's reusable `sandbox-runtime` as an opt-in), it is a substantial multi-thousand-line investment—and Section 14.4 shows the meta-harness paying the same bill a second time, one layer up.

10.3 OpenHands: Ensemble Defense-in-Depth

OPENHANDS's pluggable `SecurityAnalyzer` framework matured substantially in the V1 SDK. The Invariant analyzer of the V0 era is gone; the current set pairs an `LLMSecurityAnalyzer` (the agent self-rates each action's `security_risk`, a parameter the system prompt instructs it to fill) and a `GraySwanAnalyzer` (external adversarial API) with two new *deterministic* analyzers—`PatternSecurityAnalyzer` and `PolicyRailSecurityAnalyzer`, the latter shipping named rails such as `fetch-to-exec`, `raw-disk-op`, and `catastrophic-delete` over shell-AST parsing. An `EnsembleSecurityAnalyzer` fuses verdicts worst-case-wins, with broken child analyzers failing closed to HIGH. The LOW/MEDIUM/HIGH/UNKNOWN risk enum survives; confirmation is now a policy object (`AlwaysConfirm`/`NeverConfirm`/`ConfirmRisky`). The prompt side completes a two-sided injection defense: repository-derived context is wrapped in `<UNTRUSTED_CONTENT>` markers, and a dedicated prompt section teaches the risk-assessment protocol.

10.4 OpenClaw: Scope-Based Authorization

OPENCLAW implements scope-based authorization with `operator` and `node` connection roles over namespaced scopes (`operator.read/write/admin/pairing/...`). The gateway enforces per-method scope requirements. Additional safety mechanisms include: fixed-window rate limiting per IP/token, SSRF policy enforcement, DM allowlist with pairing approval workflow, exec approval gates for browser actions and plugin actions, and chat sanitization. The 2MB `MAX_PROMPT_BYTES` limit provides DoS protection (CWE-400).

10.5 Gemini CLI: Four Approval Modes Plus Cross-Platform Sandbox

GEMINI CLI layers two complementary safety mechanisms. First, an `ApprovalMode` enum gates every tool call into one of four modes:

- **PLAN**: Read-only; the model sees the full tool list but can only invoke read tools (Glob, Read, Grep, etc.). Its response is presented as a plan, and the user must explicitly approve before exiting plan mode.
- **DEFAULT**: Interactive approval dialog per tool call; users can approve once, approve always, or reject.
- **AUTO_EDIT**: Edit operations are auto-approved while shell and other side-effecting tools still prompt; intended for fast iteration on local code.
- **YOLO**: All actions auto-approved. Intended for CI/scripted use only.

Second, a `SandboxPolicyManager` enforces TOML-based per-mode policies: which paths are readable/writable, which network hosts are reachable, which environment variables are scrubbed. The

combination is structurally similar to CODEX's policy-as-code + sandbox stack but layers an explicit user-facing mode selector on top—an ergonomic that CODEX itself has since adopted with its collaboration-mode presets. A separate `isTrustedFolder()` check further widens or narrows context-discovery scope based on whether the directory is in the user's trust list, providing a fourth layer of opt-in granularity. The release cadence between our snapshots was dominated by policy-engine and supply-chain hardening—trust-gated `.env` loading in headless mode, shell-command validation with a core-tools allowlist, skill-install path-traversal checks—supporting Observation 6's claim that safety absorbs a disproportionate share of engineering in the largest systems.

10.6 Mistral Vibe: Permission-Scope Patterns Plus Agent-Profile Gates

MISTRAL VIBE has no OS sandbox (an opt-in `-worktree` flag now provides branch-level isolation) but compensates with a finer-grained permission model than the other no-sandbox systems. Each tool call is checked against a hierarchy:

1. Tool-wide `permission` setting (`ALWAYS`, `ASK`, `NEVER`) at the agent-profile level.
2. Tool-specific validators (e.g., `resolve_file_tool_permission` pattern-matches paths), plus a default read-only command allowlist that auto-approves `ls`/`cat`/`grep`-class commands.
3. User-declared shell hooks (`hooks.toml`, `before_tool`) that can deny a call or rewrite its inputs before permission evaluation.
4. Session rules added at runtime—now persisted across sessions for "always allow" grants.
5. An interactive approval callback presented to the user unless `bypass_tool_permissions` (the renamed `auto_approve`) is set.

Agent profiles bake in safety classifications: `plan` and `chat` profiles are `SAFE` (read-only); `default` is `NEUTRAL`; `accept-edits` is `DESTRUCTIVE`; `auto-approve` is YOLO. Tools can also override the agent default: `write_file` forces `ASK` for sensitive patterns (`.env` files) regardless of profile. There is no LLM-based risk classifier and no OS-level isolation; safety reduces to explicit per-pattern matching, hooks, and user prompts.

10.7 Hermes: A Policy Floor That Survives YOLO

HERMES inverts the sandbox question: the largest Python system in the corpus ships *zero* OS-level isolation primitives—no Seatbelt, Landlock, seccomp, or Bubblewrap anywhere in its tools tree—delegating isolation to its six pluggable execution backends. On the local backend, safety is *policy-as-code at scale*: a 3,200-line approval module whose documentation states "config.yaml IS the security policy." Its layers: user deny-globs; a twelve-pattern **hardline floor** (`rm -rf /`, `mkfs`, `dd` to block devices, fork bombs, shutdown) that survives `-yolo`—the YOLO environment variable is frozen at module import precisely so a prompt-injected skill cannot flip it at runtime; 47 dangerous-command patterns matched on *deobfuscated* variants (quote-splice removal, command-substitution folding, command-position anchoring); an external Rust content scanner (Tirith, cosign-verified install); and an optional auxiliary-LLM gate (`_smart_approve`: temperature 0, 16 max tokens, APPROVE/DENY/ESCALATE—its code credits CODEX's Smart Approvals). The distinctive threat model is *content-borne*: promptware scanning of context files, memory writes, MCP tool descriptions, and skill installs (with builtin/trusted/community trust tiers and quarantine); SSRF guards that always block cloud metadata endpoints; untrusted tool results wrapped in `<untrusted_tool_result>` delimiters with lookalike defanging. Container backends skip the dangerous-command layer entirely—until a host bind-mount re-enters it. Notable absences: no per-tool allow/ask/deny matrix, no append-only audit log.

10.8 Pi: Documented Absence as a Security Argument

PI is the corpus's only system that documents the *absence* of safety infrastructure as a design principle: its security documentation argues that a partial in-process sandbox "would be easy to

misunderstand as a security boundary," declares prompt injection via context files unprotectable, and lets built-in tools execute unconditionally with the invoking user's permissions. The one built-in gate is *project trust*: repo-controlled configuration (extensions, skills, prompts, themes) loads only from trusted paths, decided per canonical path with nearest-ancestor lookup—explicitly "not a sandbox," and context files load regardless. Permissioning is policy-as-code taken literally: the `tool_call` extension event exposes mutable tool arguments and a `{block, reason}` veto, and the reference `permission-gate` example implements the familiar regex-plus-confirm flow (`rm -rf`, `sudo`, `chmod 777`) in ∼80 lines of user space; isolation, when wanted, arrives as extensions (Anthropic's `sandbox-runtime`, a QEMU micro-VM) through the per-tool `Operations` seam. The append-only session tree doubles as a complete audit trail.

10.9 OpenCode: Syntax-Aware Permissioning, Permissive by Default

OpenCode is, after Hermes, the second-largest system shipping no OS-level isolation ("sandbox" in its code means git worktrees), and its default posture is the inverse of Codex's: `"*": allow` *inside* the project, with asks reserved for `external_directory` access, `.env` reads, and doom loops. Its distinctive investment is *syntax-aware command permissioning*: every bash command is parsed with web-tree-sitter (bash and PowerShell grammars); the exact command text becomes the ask pattern, while "always allow" grants are scoped by a ∼130-entry, explicitly LLM-generated command-arity dictionary (`git`→2, `npm run`→3), producing grants like `git commit *` rather than blanket approval; file-manipulating commands get their argument paths resolved, and out-of-project paths raise `external_directory` asks. Permission rules (`{permission, pattern, action}`, last-match-wins over glob wildcards) are layered defaults → built-in agent → user config → per-agent → per-session, and denying `"*"` for a tool removes it from the LLM's view entirely—tool availability itself is permission-derived. A parallel v2 engine inverts to deny-by-default and persists approvals to SQLite; headless runs auto-reject all asks. There is no dangerous-command denylist and no risk classifier.

10.10 Minimal Safety (Aider, Mini-SWE-Agent)

Aider offers an interactive confirmation mode but no automated risk assessment. However, its reflection loop with three-stage Python linting (syntax → compile → flake8) provides implicit safety by catching errors before they propagate.

Mini-SWE-Agent implements only resource limits as safety mechanisms—`step_limit`, `cost_limit`, and (added in the v2.4 line) a wall-clock limit and a consecutive-format-error cap, plus an opt-in interactive confirm mode—sufficient for benchmark evaluation but inadequate for production use.

11 Multi-Agent Orchestration

Nine of the eleven systems support multi-agent execution, and their architectures diverge substantially. This is the dimension on which the corpus is most spread out, so we go into more detail here than in the other sections.

11.1 Multi-Agent Taxonomy

We identify six distinct multi-agent patterns:

1. **Single-agent (no multi-agent)**: Mini-SWE-Agent, Aider—and Pi's *core*, which ships no spawn or task tool; its sub-agents live in extension space (below).
2. **Sequential delegation**: Mistral Vibe (parent spawns a sub-agent via the `task` tool and awaits completion).

3. **Parallel child sessions**: OpenHands (task and delegate tools run children as separate conversations, concurrently in threads), OpenCode (`task` creates child sessions running the same loop; multiple calls in one message execute concurrently).
4. **Hierarchical thread tree with fan-out**: Codex (dedicated threads, fork-mode-controlled inheritance, per-model tool generations, map-reduce fan-out, persisted parent/child topology).
5. **Recursive composition**: Claude Code (agents spawn sub-agents composably, with prompt-cache-sharing forks).
6. **Registry + cross-process protocol**: Gemini CLI (named agent definitions behind a symmetric local/remote session protocol, remote agents via A2A), OpenClaw (ACP session spawn), Hermes's Kanban swarm (subprocesses over a SQLite blackboard).

Table 9 provides a detailed comparison; Figure 6 gives each pattern's shape at a glance.

Table 9: Multi-agent orchestration across the nine multi-agent systems (Pi via its reference extension).

System	Mechanism	Isolation / inherit.	Communication	Depth / parallelism	Tool filtering
OpenHands	`task` + `delegate` tools → separate conversations	Fresh event log per child; metrics synced back	Summary return; conversation tree navigation	Nesting supported / concurrent threads	Markdown frontmatter (tools, skills, permission mode)
Claude Code	AgentTool + context fork	Forked context (6 dimensions), shared rendered prompt	Return + XML notifications	Recursive / true parallel	16-tool whitelist
Codex	`spawn_agent` v1/v2 (per-model)	Thread tree; `SpawnAgentForkMode`	Session input queue with mailbox phases; typed `InterAgentCommunication` records	Depth-tracked / parallel + CSV fan-out	Inherited + filtered; TOML roles (explorer, awaiter)
Gemini CLI	`invoke_agent` tool; registry defs	Isolated registries per instance	Symmetric local/remote session protocol; A2A RPC	Flat + remote / sequential in turn	Per-definition `toolConfig`
Mistral Vibe	`task` tool → new AgentLoop	Fresh config + session dir; inherits permission store, scratchpad, hooks	Event forwarding	1 hop / sequential	Subagent-tagged profiles only (explore)
Hermes	`delegate_task` in-process forks; Kanban swarm subprocesses	Fresh context, own task id; provider/creds inherited, no history	Spawn-time goal; streamed events; summary return; SQLite blackboard (swarm)	1 by default, orchestrator role unlocks nesting / 3 concurrent (pool)	Intersection with parent + 6-tool blocklist
Pi (ext.)	Extension spawns `pi -mode json -p` OS processes	Hard process isolation; fresh context	Child JSONL stdout parsed for progress/cost	Unbounded (no counter) / pool of 4, max 8; sequential `chain`	Child `-tools` flag from agent frontmatter
OpenCode	`task` tool → child Session (same loop)	`parentID` sessions; child inherits parent's deny + external-directory rules	`<task_result>` XML; background results as synthetic user messages	Recursion off by default, opt-in unbounded / parallel	Permission rulesets (deny `"*"` hides tool)
OpenClaw	ACP session spawn	Child processes; fresh session	ACP delta events	Recursive / parallel (RPC)	Session-scoped

Прикольная иллюстрация

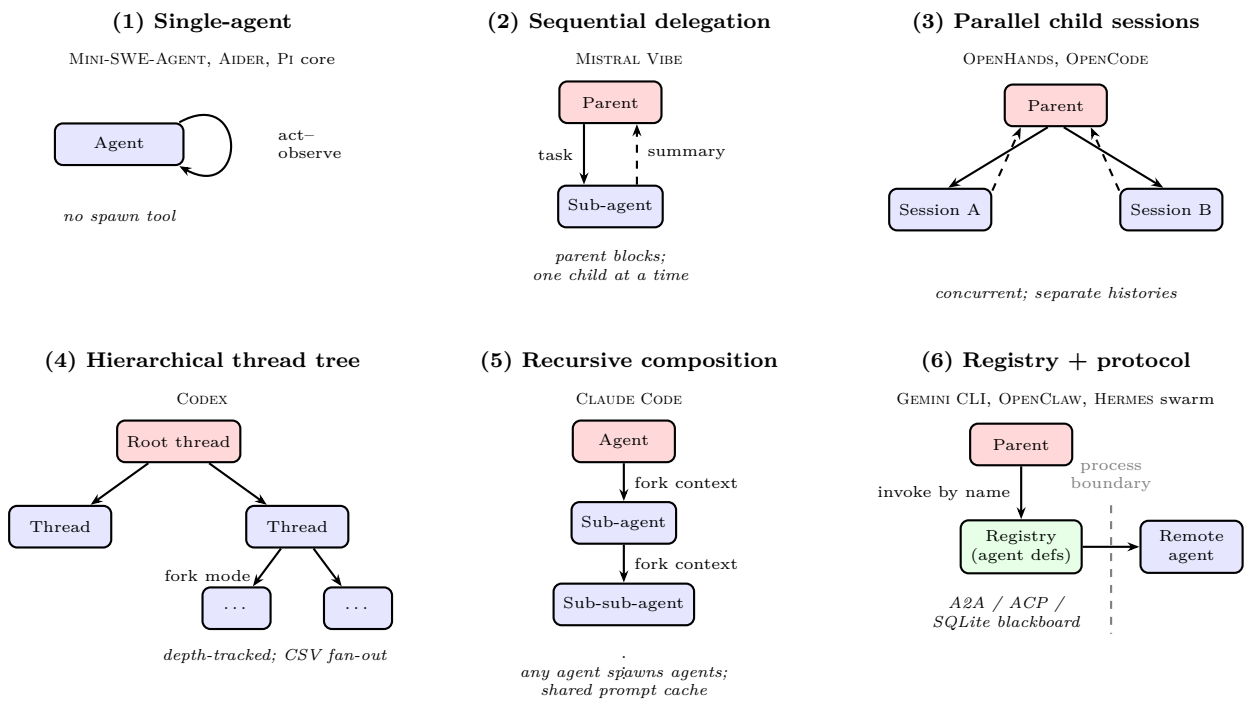


Figure 6: The six multi-agent orchestration patterns of the taxonomy above, schematically. Red nodes spawn, blue nodes are spawned, green is the definition registry; solid arrows spawn or invoke, dashed arrows return results. Systems beneath each panel are the pattern's corpus representatives (Table 9).

11.2 Deep Dive: Claude Code Recursive Composition

Claude Code's multi-agent architecture is the most sophisticated among the systems studied. It supports three levels of coordination: ad-hoc sub-agents, typed agent definitions, and full coordinator mode.

11.2.1 Agent Definitions System

Agents are defined via a JSON/Zod schema specifying a description, prompt, optional tool whitelist/blacklist, model selection, turn limits, permission mode, MCP server bindings, isolation level, background execution flag, and skills.

Agent definitions are loaded from a priority hierarchy: built-in agents → plugin agents → user settings → project settings → flag settings → policy/managed agents. Later sources override earlier ones. Built-in agents include Explore (fast codebase search), Plan (architecture design), and VerificationAgent (test execution).

11.2.2 Context Forking

When a sub-agent is spawned via `AgentTool`, the parent's context is forked along six dimensions:

1. **AbortController**: New controller linked to parent (parent abort → child abort, but not vice versa).
2. **File State Cache**: LRU cache cloned to prevent concurrent mutation.
3. **Permission Prompts**: Suppressed for async agents (`shouldAvoidPermissionPrompts = true`).
4. **App State**: Set to no-op for async agents to prevent dead session mutation.
5. **Denial Tracking**: Fresh local state (refusals don't accumulate globally).

6. **Tool Decisions**: Fresh Map per agent (no cross-agent pollution).

A critical optimization is **prompt cache sharing**: the parent's `renderedSystemPrompt` is frozen at fork time and passed verbatim to the child. This ensures prompt cache hits are maintained across fork/resume boundaries, preventing GrowthBook feature gate divergence from invalidating caches.

11.2.3 Coordinator Mode

When activated (via environment variable or feature gate), a Lead agent orchestrates Workers (Figure 7):

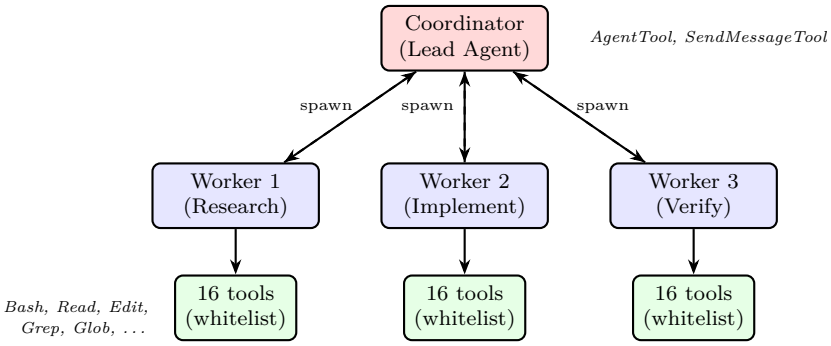


Figure 7: Claude Code coordinator mode architecture. Workers receive a whitelist of 16 tools and report via XML notifications.

Workers are restricted to a 16-tool whitelist (`ASYNC_AGENT_ALLOWED_TOOLS`): FileRead, WebSearch, TodoWrite, Grep, WebFetch, Glob, the two shell tools (Bash and PowerShell), FileEdit, FileWrite, NotebookEdit, Skill, SyntheticOutput, ToolSearch, EnterWorktree, and ExitWorktree. The coordinator synthesizes worker findings before delegating the next phase, following a structured workflow: Research → Synthesis → Implementation → Verification.

11.3 Deep Dive: Codex Thread Tree Model

Codex's multi-agent architecture is built around three core abstractions:

AgentControl is the central control plane for spawning and managing sub-agents.

AgentRegistry tracks all live agents in a session tree, maintaining `LiveAgent` metadata structs.

Spawn Flow. The `spawn_agent_internal()` function reserves a spawn slot, inherits the parent's sandbox and execution policy, resolves fork mode and metadata from the source, creates a thread with the appropriate history (full, truncated, or fresh), and emits a session-started notification.

SpawnAgentForkMode. A distinctive feature is `SpawnAgentForkMode`, which controls how much conversation history is inherited by sub-agents:

- `FullHistory`: Inherit all messages (full context).
- `LastNTurns(N)`: Inherit only the most recent N turns (reduced context).

Message filtering (`keep_forked_rollout_item()`) preserves only system/developer/user messages and final assistant answers, filtering out intermediate tool calls and reasoning. This selective inheritance prevents context bloat in sub-agents.

Inter-Agent Communication. The dedicated mailbox module of early versions has been folded into the session's input queue: inter-agent messages flow through `InputQueue` activities with mailbox delivery phases, carried as typed `InterAgentCommunication` records (`Spawn/Message/Followup/Result`). Two tool generations coexist—`multi_agents` (spawn/wait/send_message/interrupt/list) and `multi_agents_v2` (spawn/wait/send_input/resume_agent/close_agent)—selected *per model* via a `multi_agent_version` field in the model metadata, and parent/child topology is persisted by a dedicated `agent-graph-store` crate. The parent monitors child status via an `AgentStatus` enum.

Fan-Out and Declarative Roles. Beyond spawn and mailbox, CODEX has grown map-reduce-style orchestration: `spawn_agents_on_csv` launches one sub-agent per CSV row under a shared JSON-schema result contract with concurrency normalization and runtime caps, folding results back via `report_agent_job_result`. Sub-agents are typed by declarative TOML *roles* (built-in `explorer` and `worker` roles ship active in-tree; an `awaiter` role file remains but is currently unregistered) applied as high-precedence configuration layers—CODEX's answer to CLAUDE CODE's agent definitions—and delegation itself is model-governed: the `ultra` reasoning tier couples maximum reasoning with *automatic* task delegation, and configurable delegation modes (disabled / explicit-request-only / proactive) apply at thread and turn level.

11.4 OpenHands: Parallel Delegation over Conversation Trees

The V0-era `AgentDelegateAction` (sequential, one delegate at a time, shared event stream) is gone. The V1 SDK delegates through tools: a `TaskTool` whose shape is conspicuously CLAUDE CODE-like (`prompt`, `subagent_type`, `description`, `resume`), with a `TaskManager` creating a *separate* conversation per task (own event log, metrics synced back), and a `DelegateTool` with `spawn/delegate` commands that runs delegated tasks *concurrently in threads*. Sub-agents are declared as Markdown files with YAML frontmatter (name, description, model, tools, skills, hooks, MCP config, permission mode, condenser) at project/user/builtin/plugin levels—built-ins include `code-explorer`, `bash-runner`, and `web-researcher` (plus a `general-purpose` default)—and nesting is explicitly supported. Above the delegation layer sits a scaffold-level outer loop: a `/goal` endpoint runs an LLM *judge* over the transcript after each run, either injecting a follow-up prompt or stopping with a completion status, complemented by pluggable critics with configurable refinement-iteration caps.

11.5 Mistral Vibe Sequential Subagent Delegation

MISTRAL VIBE's multi-agent support is intentionally simple: a parent agent invokes the `task` tool, which constructs a fresh `AgentLoop` instance with the requested agent profile (e.g., `explore`), runs it to completion in-process, and returns the sub-agent's `AssistantEvent` outputs and `ToolResultEvent` summaries to the parent. A guard enforces that only profiles tagged as "subagent" (currently just `explore`) can be delegated to, preventing accidental recursive escalation to write-capable profiles. Each sub-agent gets a fresh `VibeConfig`, its own session directory, and an independent message history, providing isolation comparable to CODEX's thread tree but at the cost of true parallelism: the parent blocks on `await` until the sub-agent exits. The architectural simplicity (one tool, sequential, in-process) means MISTRAL VIBE sits on the *simplicity end* of the multi-agent spectrum—the corpus's simplest delegation short of none, reminiscent of OPENHANDS's retired V0 `AgentDelegateAction`.

11.6 Gemini CLI: Registry, Symmetric Session Protocol, and A2A

GEMINI CLI's multi-agent model splits across two layers. Locally, an `AgentRegistry` holds named agent definitions—Markdown files with mandatory YAML frontmatter—loaded from user, project, and extension scopes. The built-in `generalist` agent, for example, grants access to all registered tools with a 10-minute, 20-turn run budget. Invocation flows through the `/agent` slash command, `@agent-name` prompt notation, or—since the sub-agent unification of v0.39—a model-invocable `invoke_agent` tool, each run receiving an isolated tool registry. The invocation path has been

abstracted behind an *agent session protocol* with symmetric local and remote implementations, so a registry entry backed by an in-process executor and one backed by a remote A2A agent are driven through the same interface.

Where GEMINI CLI diverges from every other system in our corpus is the `packages/a2a-server` module, an experimental implementation of Google's *Agent-to-Agent* (A2A) protocol. A2A is to inter-agent communication what MCP is to tool integration: a JSON-RPC standard for one agent process to discover, authenticate to, and exchange messages with another, possibly on a different machine and built by a different vendor. GEMINI CLI's A2A server lets a remote orchestrator drive a local GEMINI CLI instance as if it were a registered local agent—now with usage metadata reported over the protocol for cross-vendor resource accounting—opening the door to cross-vendor multi-agent topologies that no other corpus system supports natively (the meta-harness of Section 14.4 builds exactly such topologies from outside).

11.7 Hermes: Guarded Delegation, Swarms, and Mixture-of-Agents

HERMES is flat-by-default delegation with two escape hatches upward. `delegate_task` builds child agents *in-process* on a thread pool: fresh conversation, focused system prompt, summary-only return, toolsets intersected with the parent's ("subagent must not gain tools the parent lacks"), and a six-tool blocklist (no recursion, no user interaction, no memory writes, no scheduling). Defaults are conservative—three concurrent children, 50-iteration budgets, depth 1—but a `role="orchestrator"` configuration plus a spawn-depth setting unlocks nested trees, with the orchestrator prompt block explicitly "modeled on OpenClaw's `buildSubagentSystemPrompt`." Background delegation returns a handle and re-enters as a *fresh turn* through a completion queue rather than splicing mid-turn—a prompt-cache-preserving choice. Above this sits a process-level tier: the *Kanban swarm* runs planning-root → parallel workers → verifier → synthesizer as separate `hermes -p <profile>` subprocesses coordinated through a SQLite board and a blackboard of structured JSON comments—coordinator-worker via shared database rather than protocol or channel. A third overlay, `/moa`, fans out advisory reference models (up to eight concurrently) whose outputs season the main model's next iteration.

11.8 Pi: Sub-Agents as an Extension, Fleets as a Package

Core PI has no spawn or task tool—sub-agents are the flagship demonstration of its extension thesis, shipped as a ∼1,000-line example. Each invocation spawns a separate `pi -mode json -p -no-session` OS process, giving hard context isolation by construction; the parent parses the child's JSONL stdout for progress and cost aggregation. One tool offers three shapes: `single`, `parallel` (worker pool of four, max eight tasks), and `chain` with `{previous}` substitution. Agent definitions are Markdown-plus-frontmatter with tool filtering via the child's `-tools` flag; there is no depth counter and no mid-run parent-child messaging. Above the single process, an experimental `orchestrator` package supervises fleets of PI instances in RPC mode and registers presence with a hosted coordinator—multi-*instance* management rather than an in-loop coordinator. Notably, when PI does do multi-agent, it lands on *cross-process JSONL* rather than in-process primitives—a data point Section 13.3 returns to.

11.9 OpenCode: Child Sessions with Permission-Derived Capabilities

OPENCODE's sub-agents are child sessions, not a separate engine: the `task` tool creates a Session with a `parentID` running the same loop, resumable via `task_id`, returning only the final assistant text wrapped in `<task_result>` XML; background results are injected as synthetic user messages. Agents and modes are unified in one schema (`mode: primary|subagent|all`): built-ins are `build`, `plan` (edit denied except plan files), `general`, read-only `explore`, plus hidden `compaction/title/summary` utility agents—plan mode is a permission ruleset plus a `<system-reminder>` injection, not an architectural overlay. Tool availability is everywhere *permission-derived* (denying `"*"` for a tool removes it from the LLM's view), and containment

is asymmetric: children inherit only the parent's deny rules. Recursion is off by default (`task` auto-denied to children) but opt-in per agent with no numeric depth cap; custom agents come from config, Markdown files, or—uniquely—*LLM generation* (`generateObject` over `{identifier, whenToUse, systemPrompt}`).

11.10 OpenClaw Session-Based Orchestration

OpenClaw's multi-agent model is fundamentally different: it operates through an ACP (Agent Client Protocol, Zed/Google's editor-oriented JSON-RPC standard at agentclientprotocol.com, not IBM's older homonymous Agent Communication Protocol) translator that bridges the Gateway wire protocol to agent processes. Sub-agents are spawned as child processes with RPC binding. Each agent gets its own session with JSONL transcript persistence. Communication occurs via ACP delta events (text, thought, tool calls) rather than shared memory. A rate limiter (`MAX_PROMPT_BYTES = 2MB`) provides DoS protection.

Observation 7. Coordinator-worker emerges independently across the corpus: in all four provider-native systems (Claude Code, Codex, Gemini CLI, Mistral Vibe), in OpenHands (parallel delegation over conversation trees), in Hermes (configuration-gated orchestrator role plus a subprocess swarm over a SQLite blackboard—a novel coordination substrate), in OpenCode (concurrent child sessions), and in OpenClaw at the protocol layer. We use coordinator-worker broadly here, for any parent-spawns-workers delegation; Section 6's narrower prescriptive overlay applies to Claude Code, Codex, and (gated) Hermes only. So many independently developed codebases converging on the same hierarchical shape is strong evidence of convergent evolution—though the convergence should be scoped to systems that build multi-agent into core: Pi is a production SWE harness that deliberately ships single-agent, relegating sub-agents to extension space. The implementations diverge instructively in what they optimize: Claude Code forks context and shares the prompt cache across children (cost); Codex builds a depth-tracked thread tree with typed inter-agent records, per-model tool generations, and CSV fan-out (isolation and scale); Gemini CLI abstracts invocation behind a symmetric local/remote session protocol (portability); Mistral Vibe runs sub-agents in-process and sequentially (simplicity); Hermes intersects toolsets and blocklists recursion by default (containment). Hadfield et al. [17] describe orchestrator-worker as the natural pattern for "valuable tasks that involve heavy parallelization, information that exceeds single context windows, and interfacing with numerous complex tools," which is a fair description of a long-horizon coding session.

11.11 Orchestration Pipelines: Claude Code vs. Codex

Claude Code and Codex are the two dominant commercial coding agents, each representing the state of the art in its provider's ecosystem. Their orchestration pipelines, end-to-end from user request to completed task, take very different design positions. Table 18 highlights the key divergences.

The table reveals two coherent philosophies whose gap narrowed measurably between our snapshots. Claude Code is *compositional-prescriptive*: it enforces structured phases (Research → Synthesis → Implementation → Verification), restricts sub-agent capabilities via tool whitelists, composes agents recursively, and optimizes aggressively around prompt-cache economics. Codex was *sandboxed-emergent*—safety guaranteed at the OS level, workflow organization left to the model—but has grown prescriptive structure of its own: a user-facing collaboration-mode selector with a Plan preset, declarative sub-agent roles, depth-tracked spawn control, and a Guardian classifier that mirrors Claude Code's middle permission layer, while Claude Code's hook vocabulary and plugin format have become Codex interfaces (Section 14.5). Both produce strong results on real-world benchmarks; the residual philosophical difference is where safety is *guaranteed* (OS enforcement vs. layered review), not how workflows are organized.

Интересно, что основное различие оказалось здесь.

12 Extensibility Mechanisms

Every system in the corpus provides extension points, though through quite different mechanisms.

12.1 Plugin Architectures

OpenClaw provides the most mature plugin architecture in the corpus—and since the spring 2026 releases, an *externalized* one: official provider and channel plugins were cut over to standalone first-class npm packages with manifest-first metadata, moving model catalogs and routing tables out of core. Plugins are manifest-driven (`openclaw.plugin.json`), discovered in the workspace directory, loaded dynamically via jiti, and registered through typed SDK contracts:

- **Channel plugins**: Implement `ChannelEntry` (receive, send, status, configure).
- **Provider plugins**: Implement `ProviderEntry` (inference stream).
- **Skill plugins**: Register via capability system.

Strict import boundaries are enforced: extension production code may only import the public plugin-SDK surface and local API barrels. Direct imports of `src/**` are prohibited.

OpenClaw is, however, no longer alone in the manifest-driven-plugin camp. Codex's plugin architecture expanded substantially between our snapshots: a marketplace-add pipeline (GitHub, git, local paths, URLs), a model-callable in-conversation approval tool (`request_plugin_install`), `.claude-plugin` format compatibility, and workspace-level sharing joined an already-present plugin manager; plugins contribute tools, skills, and lifecycle hooks. OpenHands's plugin system makes the convergence explicit: its plugins bundle skills, hooks, MCP config, agents, and commands, and its manifest loader accepts the directories `[".plugin", ".claude-plugin"]` with `plugin.json`—*it deliberately reads Claude Code's plugin format*, the clearest interoperability-by-adoption case in the corpus. Claude Code itself supports MCP servers (for external tools), skills (for specialized workflows), and agent definitions (for custom agent types) via a JSON/Zod schema with built-in, custom, and plugin sources (Section 11.2). Pi relocates the entire question: extensions are runtime-loaded TypeScript modules against an API of ∼33 typed events spanning session lifecycle, tool interception (blocking and in-place argument mutation), whole-context rewriting, and—unusually—raw provider I/O surgery (`before_provider_request`, `after_provider_response`); the repo ships 78 examples that are, feature for feature, other harnesses' built-ins, most at 1–16 KB each, distributed through a package manager (`pi install npm:...|git:...`). OpenCode's plugins are async functions returning ∼20 typed hooks (`chat.params`, `tool.execute.before/after`, `tool.definition` rewriting, multi-step OAuth `auth` flows), executed sequentially for determinism; custom tools need no plugin at all—any `.opencode/tool/*.ts` export becomes a tool.

Gemini CLI: Extensions + Slash Commands + Early MCP Adoption. Gemini CLI ships an explicit `ExtensionLoader` interface backed by the `GeminiCLIExtension` type. Extensions can contribute custom tools (typically via MCP), slash commands, hooks, skills, and `GEMINI.md` snippets, and are discovered from the `~/.gemini/extensions/` directory. Gemini CLI was an early adopter of MCP, with first-class implementations for `stdio`-spawned local servers, SSE connections, and Streamable HTTP transports, plus built-in auth providers (`google-credentials`, OAuth). MCP support extends beyond tools to *prompts* and *resources* (the broader MCP spec), which most other systems' adapters omit. Slash commands (`/agent`, `/mcp`, `/skill`, `/memory`, `/help`) are themselves extensible via skill metadata, allowing extensions to register new commands.

Mistral Vibe: Skills, MCP, Custom Tools, Agents, and Hooks. Mistral Vibe's extensibility surface has grown to five orthogonal axes. Skills follow the `agentskills.io` specification [43]: each is a directory containing a `SKILL.md` with YAML frontmatter, discovered from `.agents/skills/`, `.vibe/skills/`, `~/.vibe/skills/`, `~/.agents/skills/`, and any user-configured paths—now supplemented by built-in skills (including a "self-awareness" skill that

documents the harness to itself) and a *remote registry client* that installs versioned skills from a hosted catalog. See Section 12.5 for full cross-system comparison. MCP servers are declared as `[[mcp_servers]]` blocks supporting `http`, `streamable-http`, and `stdio` transports, with tools namespaced as `{server_name}_{tool_name}` and permissioned identically to built-ins; the layer matured between snapshots into a managed integration surface—OAuth login flows driven from the TUI (`/mcp add|login|logout|status`), a curated registry of hosted Mistral "connectors," and an *MCP sampling* handler that lets connected servers borrow Mistral Vibe's LLM backend for their own completions, a reverse dependency no other corpus system implements. Custom tools can be added via path-based discovery (`tool_paths`); custom system prompts as `~/.vibe/prompts/{prompt_id}.md`; and custom agents as `~/.vibe/agents/{agent_name}.toml` with a deep-merge override mechanism on the base config—itself now part of an explicit configuration-layer stack (default, user TOML, trusted-project `.vibe/config.toml`, `VIBE_*` environment, runtime overrides, agent profile). The fifth axis is user-facing lifecycle hooks: `before_tool/after_tool/post_agent_turn` shell hooks declared in `hooks.toml` can deny a tool call, rewrite its inputs, or append context—a direct structural sibling of Claude Code's `PreToolUse/PostToolUse`, layered on top of (not replacing) the internal middleware pipeline.

12.2 Protocol-Based Interfaces

Mini-SWE-Agent uses Python Protocols for structural subtyping: any class implementing the `Model`, `Agent`, or `Environment` protocol can be substituted without inheritance. This is the lightest-weight extensibility mechanism, requiring no registration, no manifest, and no SDK, just interface conformance.

12.3 Configuration-Driven Extensibility

Aider's model registry contains per-model metadata for 350+ models, including edit format, weak model name, cache control settings, extra API parameters, reasoning tag handling, and editor model configuration. This data-driven approach enables support for new models without code changes.

Codex's layered TOML configuration now extends *above* the user: mobile-device-management (MDM), host-wide system, and enterprise cloud-delivered layers enter as base defaults beneath user (`config.toml` plus named profiles), project (`.codex` directory), and session-flag scopes, while a separately composed requirements/constraint engine (fed from the same MDM, system, and cloud sources) validates and can hard-restrict what the lower layers may set. It is the most enterprise-governance-oriented configuration model in the corpus. Mistral Vibe now mirrors the layered approach (six layers merged by a `ConfigBuilder` with per-field merge strategies), making explicit config-layer stacks a two-system pattern.

OpenClaw's config system is the most complex in the corpus: a Zod-validated `openclaw.json` with hundreds of nested structures, plus a large auto-generated TypeScript file of validation rules for all bundled channels alongside a hand-maintained schema-help surface.

12.4 Model Context Protocol (MCP)

MCP remains a cross-cutting extension mechanism, supported by Claude Code, Codex, Gemini CLI, Mistral Vibe, OpenHands, Hermes, OpenCode, and OpenClaw—8 of 11 systems (7 of 10 coding-first). OpenHands implements MCP via `MCPToolAction/MCPToolObservation` event types, with a single `MCPServer` model validating four transports (`stdio`, `http`, `streamable-http`, `sse`) and first-class authentication up to full OAuth flows with token state. Hermes's support is near-complete (stdio/Streamable HTTP/SSE, resources, prompts, sampling, elicitation, OAuth; only prompt-list-change notifications ignored), and Hermes is *itself* an MCP server, matching OpenClaw's channel bridge. OpenCode covers tools, prompts, resources, and resource templates with OAuth dynamic client registration (sampling and elicitation explicitly disabled with tracking-issue comments). The corpus's one principled holdout is Pi, whose documentation rejects MCP

outright—"build CLI tools with READMEs"—making skills-plus-shell its articulated alternative to a wire protocol.

12.5 Skills as an Emerging Capability-Bundle Standard

Alongside MCP, a second cross-cutting extensibility mechanism has shown up in the corpus: *skills*. MCP standardizes how an agent talks to an external tool process; skills standardize how an agent packages a unit of expertise into a discoverable directory (instructions, optional scripts, optional tool whitelists, metadata) that can be dropped into a project or user home and picked up by the agent at discovery time. Mitra et al. [80] frame the same idea from the training side: skills as teachable units in agentic flows. Nine of the eleven systems implement some form of skills system; only Aider and Mini-SWE-Agent omit it (Table 10).

Table 10: Skills support across the eleven systems. "Spec" indicates conformance with the `agentskills.io` specification (`SKILL.md` + YAML frontmatter, `name`/`description` required).

System	Skills?	Discovery paths	Invocation	Spec / loading
Claude Code	✓	`.claude/skills/`, `~/.claude/skills/`, plus dynamic discovery on file edits gated by `paths` frontmatter	`SkillTool` (deferred); MCP prompts dedup'd via `uniqBy` on name	Custom (path-gated extension); hybrid eager + on-demand
Codex	✓	`core-skills/` + `skills/` roots; plugin-provided skill roots	`skills/list` app-server RPC + TUI `$`-mention; implicit-invocation detection	Custom Rust; metadata eager under a context-scaled token budget, body injected on $-mention or detected invocation; skills declare MCP/env dependencies that Codex auto-installs
Gemini CLI	✓	`~/.gemini/skills/`, `.gemini/skills/`, `.agents/skills/`, plus built-ins (`skill-creator`, `antigravity-support`); precedence workspace>user>built-in	`ActivateSkill` tool wraps skill in XML with `available_resources` tree; permission-confirmation UI	agentskills.io-aligned with `isBuiltin` extension; eager startup walk; path-traversal-hardened installs
Mistral Vibe	✓	`.agents/skills/`, `.vibe/skills/`, `~/.vibe/skills/`, `~/.agents/skills/`, user `skill_paths`; trust-gated project discovery	`skill` tool (deferred); user `/skill-name` invocations materialized as synthetic tool calls	Full `agentskills.io` compliance; remote registry client (versioned catalog); builtin self-awareness skill
OpenHands	✓	`{workdir,git-root}/.agents/skills/`, `.openhands/skills/` (+legacy microagents); user equivalents + managed installs	Metadata in `<available_skills>`; body on demand via builtin `invoke_skill` tool	agentskills.io flag; *progressive disclosure*; `PathTrigger` rules on file-touch; server CRUD/sync/marketplace API; ingests AGENTS.md/.cursorrules as scoped rules
Hermes	✓	`~/.hermes/skills/` (seeded from 72 bundled; 102 more official skills installable via the Skills Hub) + read-only `external_dirs`	Three-tier progressive disclosure: index in prompt → `skill_view` → linked assets; skills double as slash commands	agentskills.io-compatible; trust-tiered Skills Hub (builtin/trusted/community) with pre-install scanning and quarantine; self-authoring via `skill_manage`
Pi	✓	`~/.pi/agent/skills/`, `~/.agents/skills/`, `.pi/skills/`, `.agents/skills/` (cwd to git root), packages, `-skill`; project paths trust-gated	No dedicated tool: `<available_skills>` XML index; body loaded via the ordinary `read` tool; `/skill:name` expansion	agentskills.io, deliberately lenient (warns, still loads); `disable-model-invocation` gating
OpenCode	✓	`{skill,skills}/**/SKILL.md` in config dirs, `~/.claude/skills`, `.claude/`, `.agents/` walk-up, extra paths, remote URL registries (versioned `index.json` cache)	Native `skill` tool (deferred: catalog lists name+description only)	agentskills.io field set; per-skill permission gating
OpenClaw	✓	`~/.openclaw/skills/`, `.agents/skills/`, plugin-provided, `skills.load.extraDirs` config	Injected as XML in system prompt; `/skills` slash command lists eligible	agentskills.io-compatible; gating via `metadata.openclaw.requires` (`bins`, `env`, OS); governed installs (Skill Workshop approval flow, provenance-verified ClawHub)
Aider	✗	—	—	—
Mini-SWE-Agent	✗	—	—	—

Convergence on the SKILL.md format and the `.agents/skills/` path. The format is essentially universal among the nine adopters: a directory named after the skill, containing a `SKILL.md` file with YAML frontmatter (at minimum a `name` and `description`, often with optional fields like `paths`, `requires`, or `user-invocable`). The `.agents/skills/` discovery path,

the canonical agentskills.io location, is accepted by six systems (Mistral Vibe, Gemini CLI, OpenClaw, OpenHands, Pi, OpenCode), and the vendor-specific homes (`~/.claude/skills/`, `~/.gemini/skills/`, `~/.vibe/skills/`, `~/.hermes/skills/`, `~/.pi/agent/skills/`, `~/.openclaw/skills/`) coexist with project-local equivalents. The sharpest interoperability datum: OpenCode *deliberately searches a competitor's home*—its discovery list includes `~/.claude/skills` and `.claude/` directories, so skills installed for Claude Code work in OpenCode unmodified.

Skills versus MCP: the tie has broken. MCP and skills address different layers. MCP is a *wire protocol* for an agent to talk to a separately running process that exposes tools, prompts, and resources; skills are a *file-system convention* for packaging an instruction-plus-script bundle the agent reads directly. The two compose: Claude Code's skill registry deduplicates MCP-supplied prompts that look like skills, and Mistral Vibe composes in the other direction, letting MCP servers borrow its LLM via sampling while skills arrive from a hosted registry. In April the two standards were tied at 6/8 adoption; the expanded July corpus breaks the tie in skills' favor—**skills 9/11, MCP 8/11**—because Pi implements agentskills.io while rejecting MCP outright. Skills-as-MCP-replacement is now an articulated position, not a coexistence.

Deferred loading as the dominant pattern. Echoing Claude Code's deferred tool loading (Section 8.3), eight of the nine adopters now load only skill *metadata* eagerly and fetch bodies on demand—via a dedicated tool (`SkillTool`, `ActivateSkill`, `skill`, OpenHands's `invoke_skill`), via $-mention and implicit-invocation detection (Codex), via a three-tier disclosure chain (Hermes), or, most minimally, via the ordinary `read` tool against an XML index (Pi). OpenHands, an eager outlier in April, moved to progressive disclosure with the V1 SDK; OpenClaw remains the one eager loader, compensating with eligibility gating so ineligible skills never reach the prompt.

Gating and conditional activation. The most architecturally interesting variation is conditional activation. OpenClaw's skills declare runtime requirements (binaries on `PATH`, environment variables, OS family) and are filtered at load time, preventing "how to use `kubectl`" from polluting the prompt on a machine without it. Claude Code's `paths` frontmatter activates a skill when the model touches matching files (via the Read, Edit, or Write tools), and OpenHands now implements the same idea as `PathTrigger` rules injected on file-touch—marrying skills to the JIT context philosophy of [19]. This is a structural advance over MCP, which treats every connected server symmetrically regardless of whether its tools are relevant to the current task.

Skills grow a supply chain—and authors. Distribution has professionalized in one quarter. Remote registries now exist in four systems (Mistral Vibe's hosted catalog client, OpenCode's URL registries with versioned index caches, Hermes's Skills Hub, OpenHands's marketplace API), and with distribution comes supply-chain security: Hermes applies trust tiers, pre-install static scanning, quarantine, and dependency checks; OpenClaw routes installs through an approval workflow with provenance-verified sources; Gemini CLI hardened skill installs against path traversal; Codex auto-installs a skill's declared MCP-server dependencies (optionally after prompting). More striking still, skills have acquired *non-human authors*: Hermes's background-review agent creates and patches skills from completed tasks (with a curator maintaining the collection), and Gemini CLI's extraction sub-agent mines sessions into skill patches for inbox review. The capability-package layer is acquiring package-manager economics—registries, provenance, and now generated packages—at remarkable speed.

Observation 8. Skills have overtaken MCP as the corpus's most-adopted extensibility standard: 9/11 systems implement SKILL.md bundles (only Aider and Mini-SWE-Agent abstain) versus 8/11 for MCP, the tie of April broken by Pi's explicit skills-yes-MCP-no position. Three second-order developments mark the layer's maturation: deferred loading is now the near-universal strategy (8 of 9 adopters); conditional activation (Claude Code `paths`, OpenHands `PathTrigger`, OpenClaw `requires`) pushes JIT context engineering into the extensibility layer; and a supply chain has emerged—hosted registries in four systems, trust tiers and quarantine in Hermes, provenance verification in OpenClaw—together with the first agent-*authored* skills (Hermes's self-improving loop, Gemini CLI's extraction inbox).

13 Cross-Cutting Observations and Implications

13.1 Architectural Pattern Catalog

Tables 11 and 12 catalog the 29 recurring design patterns identified across the eleven systems: the seventeen of the April edition (with membership updated) plus twelve contributed or crystallized by the expanded corpus.

Table 11: Catalog of 29 recurring architectural patterns (1/2): the April-edition seventeen, membership updated to July 2026.

Pattern	Description	Used By
Event Sourcing	Actions/observations appended to a persistent event log	OpenHands, Pi (session log-as-tree), OpenCode (log-as-queue; v2 tables)
Policy-as-Code	Safety rules as executable configurations	Codex (Starlark, with inline validated examples), Claude Code (hooks), Gemini CLI (TOML modes), Hermes (config-as-policy + hardline floor), OpenCode (rulesets), Pi (extension hooks)
Recursive Composition	Agents spawn sub-agents with forked/linked context	Claude Code, Codex, OpenHands, OpenClaw; config-gated in Hermes, opt-in in OpenCode
Polymorphic Edits	Model-aware edit format/toolset selection	Aider (prompt factory), OpenCode (tool registry)
Deferred Loading	Tools/skills hidden from prompt, discovered on demand	Claude Code, Codex (BM25 `tool_search`), Hermes (BM25 bridge tools), + 8 skills implementations
Template Method	Base class defines flow; subclasses override parse/format	Aider (Coder), OpenHands (Agent)
Protocol Interfaces	Structural subtyping / interface seams for pluggable components	Mini-SWE-Agent, Pi (per-tool `Operations` remoting seam)
LLM Summarization	LLM compresses conversation history	9 systems (all but Mini-SWE-Agent, Aider-partial)
Stuck Detection	Automated detection of repetitive agent behavior	OpenHands (5 scenarios), Gemini CLI (hybrid hash+LLM), Hermes (warn-first signatures), OpenCode (doom-loop ask), Mini-SWE-Agent (format-error cap)
Reflection Loop	Inner self-correction cycle with lint/test feedback	Aider; cousins: Gemini CLI (edit fixer), OpenCode (LSP feedback)
Prompt Caching	Cache-boundary prompt structure or session-key reuse	Claude Code, Codex, OpenHands (cache tiers), Hermes (prefix normalization), Pi (breakpoints+TTL), OpenCode (dialect fanout)
Context Forking	Cloning parent state for sub-agent isolation	Claude Code, Codex, Hermes (cache-sharing background fork)
Middleware Pipeline	Composable turn-level policies orthogonal to loop body	Mistral Vibe
JIT Repo Context	Hierarchical Markdown context files auto-discovered + on-demand injection	Claude Code, Codex, Gemini CLI, Mistral Vibe, Hermes, Pi, OpenCode, OpenHands
Skills (capability bundles)	SKILL.md directories with YAML frontmatter	9 systems (all but Aider, Mini-SWE-Agent)
Conditional Activation	Skills/tools gated by environment or file paths	Claude Code (`paths`), OpenHands (`PathTrigger`), OpenClaw (`requires`)
Turn-Level Checkpoint	Filesystem snapshots with rewind and file restoration	Mistral Vibe (per user message), OpenCode (shadow-git, per step), Hermes (shadow-git store), Pi (conversation-only)

Table 12: Catalog of 29 recurring architectural patterns (2/2): the twelve patterns new to the July corpus.

Pattern	Description	Used By
Agent-Maintained Memory	Background sub-agent extracts and consolidates cross-session memory	Codex (two-phase, git-baselined); human-gated variant in Gemini CLI (patch inbox)
Outer Verification Loop	Scaffold-level judge/guard validating completion, outside the turn loop	OpenHands (`/goal` judge + critics), Hermes (verify-on-stop)
Self-Improving Skill Loop	Agent authors, patches, and curates its own capability bundles	Hermes; partial: Gemini CLI (skill-extraction inbox)
Lineage Compaction	Compaction as session rotation with searchable ancestry chain	Hermes
Session-Tree Version Control	Append-only entry tree with movable head; fork/rewind/branch summaries	Pi; OpenHands (conversation tree)
Minimal-Core / Extension-Host	Safety, sandbox, sub-agents, plan mode relocated to a runtime event bus	Pi
Client/Server Harness	Embedded API server; every UI (TUI, web, IDE, CI) is a client	OpenCode; OpenHands (agent-server)
Model-Family Prompt Matrix	Distinct base prompts dispatched per model family/generation	Codex (server catalog), OpenCode (9 prompts), Hermes (gated blocks)
Cache-Dialect Fanout	All providers' cache-control dialects emitted simultaneously	OpenCode
Syntax-Aware Command Permissioning	Commands parsed (tree-sitter) and grants scoped by argument arity	OpenCode; Mistral Vibe (parse-validation), Hermes (deobfuscated matching)
Untrusted-Content Delimiting	Tool/web results wrapped in taint markers with delimiter defanging	Hermes, OpenHands (`<UNTRUSTED_CONTENT>`)
Harness Mimicry	Client presents a first-party harness's identity (headers, prompt opening, tool-name casing) to ride its subscription OAuth backend	Pi (Claude Code identity on Anthropic OAuth; ChatGPT-plan Codex backend)

13.2 The Twin Absences: No Agentic Frameworks, No Code RAG

Two technologies that the broader LLM-application literature treats as central to any production agent turn out to be missing from all eleven systems.[8] For this analysis, the absences are at least as informative as the presences.

Absence 1: agentic frameworks. Every dependency manifest was inspected, and every source tree grepped for imports of the widely deployed agentic frameworks: LangChain, LangGraph, LlamaIndex, AutoGen, CrewAI, Pydantic AI, Genkit, Haystack agents, Semantic Kernel, Google's ADK, and several smaller libraries (Smolagents, Swarm, Agno). Across roughly 4 M lines of Python, TypeScript, and Rust, no production agent code path imports any of them—a result worth savoring in two directions: Gemini CLI uses neither of Google's own frameworks (Genkit, ADK), and Hermes's only "LangChain" strings sit inside bundled skill *documentation* teaching the agent how users might use vector databases. Two boundary cases deserve precision. Aider ships one optional framework-adjacent extra: its `/help` command can install `llama-index` to run doc-RAG over *aider's own documentation*—an opt-in feature dependency, not agent-loop orchestration. And OpenCode delegates its inner LLM/tool plumbing to Vercel's AI SDK—a provider-abstraction layer, not an agent-orchestration framework, but the first corpus system whose inner loop plumbing is a third-party SDK at all (an in-house replacement client sits behind a flag, suggesting the dependency is transitional).

[8] We expected to find at least some use of LangChain or another framework in the open-source projects. The uniformity surprised us; the search for counterexamples (vendored dependencies, dynamic imports, transpiled TypeScript builds) ran for several weeks before we accepted the result. The July re-audit repeated the manifest and import sweep across all twelve trees, including the three systems new to the corpus and the meta-harness.

Every loop is hand-rolled in the host language's native primitives: `asyncio` (OpenHands, Mistral Vibe, Hermes), blocking synchronous Python (Aider, Mini-SWE-Agent), `Promise`/async-iterator (Claude Code, Gemini CLI, Pi, OpenCode, OpenClaw), Tokio (Codex). Every tool registry is custom-built around Pydantic, Zod, TypeBox, Effect Schema, or Rust enums. Every prompt template is plain Markdown, Jinja2, or string concatenation.

Absence 2: retrieval-augmented generation over code. A parallel search covered vector-store dependencies (Chroma, Pinecone, Weaviate, Qdrant, Milvus, FAISS, LanceDB, sqlite-vec, Elasticsearch in vector mode), embedding libraries, and folders or files matching `embedding`, `vector_store`, `vectordb`, `rag`, or `retrieval`. For *code* retrieval the result is unchanged and now spans eleven systems: zero. The exceptions all concern *conversation memory*, and here the July picture did shift: OpenClaw's default memory plugin (`memory-core`) now runs a hybrid sqlite-vec KNN + FTS5/BM25 search with embeddings on by default (the provider defaults to OpenAI; a local GGUF model is opt-in; the April-era LanceDB extension survives as an optional plugin), making OpenClaw the one system where embeddings are on by default—for chat recall, never for reading the source tree. Hermes demonstrates the opposite choice at scale: its core past-conversation search is deliberately lexical (trigger-maintained SQLite FTS5, BM25 plus trigram tokenization for CJK, "no LLM calls anywhere"), with embeddings confined to opt-in memory plugins.

What the eleven systems use *instead* of RAG for code retrieval is summarized in Table 13.

Table 13: Code-retrieval mechanisms used by each system, in lieu of vector-based RAG (July 2026).

System	**Embeddings used?**	**Code-retrieval mechanism**
OpenHands	No	`GrepTool`/`GlobTool` + terminal; context-file ingestion (AGENTS.md, .cursorrules); tree-sitter no longer present in the SDK
Aider	Optional `/help` extra only (doc-RAG over aider's own docs)	`RepoMap`: tree-sitter symbol extraction with token-budget-constrained PageRank-style ranking
Claude Code	No	ripgrep keyword search + BashTool + on-demand file Read; CLAUDE.md auto-discovery
Codex	No	Rust-native file search; AGENTS.md root-to-cwd concatenation
Gemini CLI	No	Bundled ripgrep + glob tools; GEMINI.md auto-discovery; `MEMORY.md` project index; JIT subdirectory context
Mistral Vibe	No	ripgrep + tree-sitter (bash parsing) + filesystem search; git status injection; JIT nested AGENTS.md
Mini-SWE-Agent	No	Grep-based search via shell tools
Hermes	Opt-in memory plugins only	ripgrep-backed `search_files`; SQLite FTS5 (BM25 + trigram) over *session history*; LSP for post-write diagnostics only; no repo map
Pi	No	Auto-downloaded ripgrep + fd binaries (latest release; system binaries preferred) behind `grep`/`find` tools; ancestor-walk context files; session search is a deterministic linear scan
OpenCode	No	Bundled ripgrep (`grep`/`glob`, 100-result cap); ∼25 auto-downloaded LSP servers feed diagnostics (no persistent index); lazy nested AGENTS.md attach
OpenClaw	**Default** hybrid memory search (sqlite-vec KNN + FTS5/BM25; OpenAI embedder by default, local GGUF opt-in) for *conversation memory only*, never code	N/A for code retrieval; standard filesystem APIs

Why the twin absences matter. The first absence corroborates advice from the agent designers themselves. Schluntz & Zhang's *Building Effective Agents* [16] cautions that frameworks "often create extra layers of abstraction that can obscure the underlying prompts and responses, making them harder to debug," and recommends starting from raw SDK calls. That advice appeared in December 2024; the source-code audit we run here, in 2026, shows that every commercial provider has followed it. This is worth noting because the broader Python LLM-application community has kept investing in framework abstractions over the same period. Production SWE agents appear to operate on a different complexity budget from generic LLM applications. Hand-rolled debuggable code wins out over reusable abstractions once the agent is mutating real source code, because the failure modes (silent prompt corruption, opaque caching, version-incompatible tool schemas) become too expensive to shrug off.

The second absence is consistent with recent guidance from Rajasekaran et al. [19] (September 2025), who favor "just-in-time" (JIT) retrieval while recommending a hybrid approach: "We don't memorize entire corpuses of information, but rather introduce external organization and indexing systems like file systems, inboxes, and bookmarks to retrieve relevant information on demand." The blog post does not flatly reject pre-indexed retrieval, but its practical recommendations all point toward JIT methods. Every system in the corpus follows this approach, even though repository-level retrieval over code is a well-studied area: RepoCoder [60], RepoBench [61], and CrossCodeEval [62] all build retrieval pipelines for code, and Long Code Arena [64] provides matching long-context evaluation. Wang et al.'s CodeRAG-Bench [63] asks the very question this section raises, "can retrieval augment code generation?", and finds gains that are highly variable across tasks: substantial for documentation-lookup and library-use scenarios but marginal or absent for tasks where the model already has sufficient parametric knowledge. The inconsistency of these gains may help explain why production SWE agents skip the RAG layer entirely rather than investing in task-dependent retrieval routing. The reasons are domain-specific. Code carries dense deterministic structural metadata—file paths, language servers, tree-sitter parses, type information—that semantic-similarity retrieval cannot replicate (and that structure-aware tools such as RepoAgent [65] and Aider's `RepoMap` exploit directly). Code changes minute to minute, so pre-indexed embeddings are stale almost by construction. Every coding environment already ships a near-optimal retrieval system in the form of `ripgrep`, `find`, and `glob`. On the typical SWE-agent task, RAG adds operational cost (embedding compute, vector-store maintenance, drift management) without offering marginal value.

Observation 9. Two technologies that the broader LLM-application literature treats as central are missing from all eleven systems—a finding that survived a threefold corpus expansion and a three-month re-audit. No system uses a general-purpose agentic framework in its agent runtime (LangChain, LangGraph, AutoGen, CrewAI, or any of a dozen others checked; Gemini CLI uses neither of Google's own); every loop is hand-rolled in the host language's async primitives, with OpenCode's use of a provider-abstraction SDK for inner plumbing as the closest boundary case. No system uses vector-embedding RAG for code retrieval; all rely on `ripgrep`, tree-sitter, glob, and auto-discovered Markdown context files, and where conversation-scale recall is needed, the production answer is lexical search (Hermes's SQLite FTS5) or, in exactly one default configuration (OpenClaw), hybrid embeddings over chat history—never over the source tree. Production harnesses operate on a different complexity budget from generic LLM applications: debuggability and prompt transparency outweigh framework reuse when the failure mode is mutating real code. Section 14.2 gives the absence its historical resolution.

Observation 10. The architectural patterns described in Anthropic's *Effective Agents* engineering series (December 2024 through September 2025) line up closely with the architectures observed across the four provider-native systems, which were built independently. Whether this reflects shared empirical reality, public-guidance influence, or both is an open question.

13.3 Inter-Agent Protocols: Outward Adoption, Inward In-Process

The twin absences are about two technologies that simply do not appear in the corpus. Inter-agent protocols (ACP, A2A) follow a different pattern again—and this is the dimension that moved most between our snapshots. ACP now ships as a first-class production dependency or implementation in *six* of the eleven systems: Mistral Vibe (`agent-client-protocol==0.10.1`, with session fork, workspace trust, and rewind exposed over the protocol), OpenClaw (gateway translator), OpenCode (`opencode acp` serves the agent side over stdio), Hermes (ACP adapter and registry modules), OpenHands (an `ACPAgent` class, discussed below), and Gemini CLI (whose A2A module is a separate protocol for the mesh role). Outside the corpus, xAI's Grok Build launched with a documented ACP server mode on day one (Section 3.7; vendor-documented, not source-verified). None of the corpus adoptions is experimental.

What is interesting is the placement—and ACP now occupies *three distinct architectural roles*:

1. **Editor ↔ agent (outward server).** The LSP role for ACP [41]: Zed, JetBrains, and other IDEs drive a local agent as they would a language server. Mistral Vibe, OpenCode, Hermes, OpenClaw, and Grok Build all serve this boundary.
2. **Agent-as-backend (inward host).** The role nobody held in April: OpenHands's `ACPAgent` delegates its `step()` to an external ACP server, with provider metadata for pinned `claude-agent-acp`, `codex-acp`, and `gemini -acp` binaries—*rival harnesses become interchangeable brains inside an OpenHands conversation.* The meta-harness of Section 14.4 drives its Goose and Qwen adapters the same way (and taps Kiro's ACP permission stream while driving its TUI), and Hermes consumes an ACP agent as a *model transport* (GitHub Copilot's CLI as a chat backend). The protocol built for editors turned out to be the interface for hosting.
3. **Cross-vendor mesh (A2A [42]).** Still Gemini CLI's alone in the corpus: a remote orchestrator drives a local Gemini CLI as one node in a multi-vendor topology, now with usage metadata over the wire.

Table 14 lists the communication mechanism each multi-agent system uses between a main agent and its own sub-agents, alongside its protocol roles.

Table 14: Main-agent ↔ sub-agent communication and protocol placement (July 2026).

System	Main ↔ sub-agent	Protocol roles	Sub-agents cross-proc.?
Claude Code	`AgentTool` function call + context fork + XML notifications	None at agent layer (ACP served by a separate adapter binary)	No
Codex	Session input queue with mailbox phases; typed records	None at agent layer (`codex-acp` adapter external)	No
Gemini CLI	`invoke_agent` behind local/remote session protocol	**A2A** server (mesh role)	No (internal)
Mistral Vibe	`task` tool, in-process `asyncio`	**ACP** server (editor role; rewind over protocol)	No
OpenHands	Task/delegate tools, concurrent threads	**ACP** *client/host*: rival harnesses as step() backends	No (own sub-agents)
Hermes	In-process thread forks; swarm via SQLite blackboard subprocesses	**ACP** server (editors) *and* client (Copilot CLI as model backend)	Swarm: yes (DB, not protocol)
Pi	Extension spawns OS processes, JSONL over stdio	Proprietary JSONL RPC (∼30 commands) for embedding; ACP/A2A/MCP absent by design	**Yes** (ext.; JSONL, not ACP)
OpenCode	`task` tool, in-process child sessions	**ACP** server (agent side, stdio ndjson)	No
OpenClaw	**ACP session spawn** over RPC (child process)	Same ACP inward and outward	**Yes**

The inward/outward split of the April edition survives in refined form. For coordinating *their own* sub-agents, eight of the nine multi-agent systems still use in-process primitives or, where they cross processes, something other than the standard protocols: Pi's extension spawns OS processes speaking plain JSONL, and Hermes's swarm coordinates through a SQLite blackboard. What changed is the *inward consumption of whole harnesses*: hosting a rival agent as a swappable backend—OpenHands's ACP hosting, Hermes's ACP-as-model-transport, the meta-harness's adapter fleet—is a production pattern that did not exist in April, and ACP is its lingua franca. Pi adds a final nuance: it rejects the standard protocols yet *invents* a proprietary outward RPC for the same embedding role—evidence that the outward-facing protocol pressure is real even where the standards are declined.

Observation 11. Inter-agent protocol placement has evolved from a two-role story (outward: editor integration and mesh; inward: nothing) to a three-role story. ACP ships in six of eleven systems and now serves (1) its designed editor↔agent boundary, (2) a role outside the protocol's design brief—*harness hosting*, where OpenHands runs Claude Code, Codex, or Gemini CLI as interchangeable step-backends and Hermes consumes an ACP agent as a model transport—and (3) via A2A, still only in Gemini CLI, the cross-vendor mesh. For a harness's *own* sub-agents, eight of nine multi-agent systems still use in-process primitives or, where they cross processes, something other than the standard protocols (Pi's JSONL extension, Hermes's SQLite-blackboard swarm); OpenClaw's ACP spawn is the lone case that routes sub-agent coordination over a standard protocol. The practitioner guidance sharpens accordingly: build an ACP *server* (it now buys you editors, hosts, and meta-orchestrators at once); keep your sub-agents in-process; and treat A2A as a bet on a cross-vendor mesh whose at-scale demand remains unproven.

13.4 Trade-off Framework

Люблю когда так прямо подсвечивают компромиссы :)

We formalize five fundamental trade-off axes:

Axis 1: Simplicity vs. Capability. Mini-SWE-Agent's minimal scaffold reportedly achieves 74%+ on SWE-Bench Verified, while Codex's far larger codebase reports 69.1%. These figures are not directly comparable (different underlying models, different evaluation runs, different deployment configurations: Mini-SWE-Agent is unsandboxed and Codex is sandboxed) and we do not draw a head-to-head conclusion from them. What the gap does illustrate, qualitatively, is that the additional code in Codex is not invested in raw task-completion logic: a substantial share goes into safety (cross-platform sandboxing), user experience (TUI, streaming), extensibility (MCP, plugins), and robustness. Whether the production scaffolding produces measurable improvements on the same benchmark under matched conditions is an open question that this study does not answer (see Section 15 on threats to validity).

Axis 2: Safety vs. Autonomy. Systems with more safety infrastructure (Codex, Claude Code) impose more friction on agent actions. Systems with less safety (Mini-SWE-Agent, Aider) enable faster execution but are unsuitable for enterprise deployment without additional safeguards.

Axis 3: Provider Coupling vs. Agnosticism. Claude Code and Codex exploit provider-specific features (prompt caching, extended thinking, model-specific prompts) at the cost of vendor lock-in. LiteLLM-based systems sacrifice these optimizations for provider flexibility.

Axis 4: Monolithic vs. Modular. Codex's 126-crate, million-line Rust workspace optimizes for type safety and performance. Mini-SWE-Agent's single-file agent maximizes comprehensibility. Pi relocates the entire axis into runtime composition: a minimal core plus an extension event bus. OpenClaw's plugin SDK and OpenCode's client/server split enable extensibility and embeddability respectively. Each point on this spectrum serves different user populations.

Или Pi и нравится тем, кто собирает своих автономных агентов

Axis 5: Scaffold Complexity vs. Model Capability. This is the most philosophically interesting axis. Mini-SWE-Agent's competitive benchmark performance demonstrates that current models are capable enough to succeed with minimal scaffolding. Production systems invest in scaffolding not because models need it for task completion, but because users need it for safety, reliability, and workflow integration.

Интересный тезис, но он не бесспорный
Вероятно, SWE Bench просто уже содержит слишком простые задачи :)
Но, вообще, тенденция мерить не только success rate есть.

Интересное наблюдение и довольно точное

14 The Platform Turn

The preceding sections dissected the harness as an artifact. This section argues that the artifact has changed category: between 2025 and mid-2026, the coding-agent harness completed a turn from *tool* to *platform*—a runtime with its own extension ecosystem, package economics, governance layers, switching costs, and, since June 2026, its own meta-layer. The April edition of this study advanced a cautious version of this claim as the "CLI-as-framework hypothesis." The evidence accumulated since—in the corpus's own source trees, in the vendor SDKs, and in the market—lets us state it as a thesis.

14.1 From Hypothesis to Thesis: The CLI as Framework

The standard reading of the absence of LangChain, AutoGen, and the rest (Observation 9) is that production harnesses do not *need* frameworks. But there is a second reading: they do not need frameworks because the harness *is itself the framework*—the default orchestration layer through which developers ship software, not merely a tool they query. ← Обвязки уже сами стали фрейворк.

Consider what a framework traditionally provides: a loop, a tool registry, a memory layer, a configuration surface, an extension mechanism. Every one of these is present in every production system studied here. Claude Code ships a streaming ReAct loop, 43 registered tools, a compaction-based memory manager, CLAUDE.md-driven configuration, MCP for integrations and Skills for capability bundles—and Codex, Gemini CLI, Mistral Vibe, Hermes, and OpenCode do the same, each in its own idiom (Pi matches every element but MCP, which it pointedly rejects in favor of CLI tools plus its extension bus). The difference is that the "framework" is not a library the developer imports into their code; it is a runtime the developer works *inside of*. The developer's "program" is a natural-language task plus a tree of Markdown files (CLAUDE.md, AGENTS.md, GEMINI.md, SKILL.md) that configure the agent's behavior, tool access, and domain knowledge. The "API" is the filesystem, the terminal, and git.

Four convergent signals from the corpus support this reading.

Signal 1: Skills as declarative programs. Nine of the eleven systems implement Skills (Observation 8): directories containing a `SKILL.md` with instructions, optional scripts, and metadata. A skill is, functionally, a program written for an LLM runtime rather than for a CPU runtime. It declares what the agent should do (`SKILL.md` body), when it should activate (`paths` frontmatter in Claude Code, `PathTrigger` rules in OpenHands, `requires` in OpenClaw), and what tools it may use. This is the structural analogue of a plugin in a traditional framework, except the programming language is English plus YAML frontmatter—and the layer now has registries, supply-chain security, and agent authors (Section 12.5).

Signal 2: Hooks and event buses as the extension substrate. In April, user-facing lifecycle hooks were a Claude Code distinctive. By July they are the dominant extension substrate—nine of the eleven systems, with only Aider and Mini-SWE-Agent abstaining: Codex ships hooks whose event names are *verbatim* Claude Code's; OpenHands wires `pre_tool_use`/`post_tool_use`/`stop` hooks into `Agent.step()`, with an *agent* allowed as a hook handler; Mistral Vibe added `hooks.toml` shell hooks (deny, rewrite inputs, append context) alongside its internal middleware pipeline; Gemini CLI exposes eleven loop-lifecycle events to extensions; Pi *is* an event bus, with ∼33 typed events spanning tool interception to raw provider I/O; OpenCode's plugins are functions returning twenty typed hooks. This is the same extension model that web frameworks and build systems have used for decades, applied to an LLM execution loop. The developer is not writing *for* the agent; they are writing *inside* it.

Signal 3: The disappearing boundary between tool and workflow. In a traditional development workflow, the developer writes code, runs tests, reads logs, commits, and opens PRs as discrete steps. In a harness workflow, these steps are delegated to the agent via natural-language

instructions and executed through the agent's tool system. The agent loop *is* the workflow engine: Claude Code's coordinator phases, Codex's `update_plan` tool and `/goal` workflows, OpenHands's judge-audited goal loop. When the developer types a task and the agent spawns sub-agents, dispatches tools, manages context, and produces the result, the CLI has subsumed the role of the build system, the task runner, and the IDE.

Signal 4: The harness as a service surface. The clearest post-April signal: harnesses now ship the interfaces of platforms. OpenCode embeds an HTTP server publishing an OpenAPI spec and a generated SDK, with every UI—TUI, desktop, web, IDE plugin, CI action—as a client. OpenHands exposes conversations through an OpenAI-*compatible* gateway, so any tooling that can call a chat-completions endpoint can drive an agent: the agent as a model. Mistral Vibe teleports local sessions to a hosted runtime; Claude Code pairs its SDK with a hosted Managed Agents API. A runtime with clients, SDKs, and hosted tiers is not a tool with an ecosystem; it is a platform with distribution.

14.2 The Harness–Framework Merger

If the harness is the framework, the two artifact categories should merge—and in 2026 they are merging, visibly, from both directions (Table 15).

Table 15: The harness–framework merger, July 2026: named, versioned, installable artifacts.

Direction	Artifact	What it is
Harness → framework	Claude Agent SDK [38]	"Claude Code as a library": the same tools, loop, context management, hooks, and subagents, importable in Python/TypeScript; plus a hosted Managed Agents API
	`openai-codex` SDK	`pip install openai-codex`: start Codex threads, run turns, stream progress, control workspace access; Python/TypeScript, shipped in the Codex repo
	OpenHands agent SDK	The V1 restructuring *is* an SDK-ization: the agent became `openhands-sdk`/`openhands-tools` packages plus an agent server
	Pi packages / OpenCode SDK	`pi-agent-core` importable + RPC mode; generated `@opencode-ai/sdk` over the embedded server
Framework → harness	Deep Agents [37]	LangChain's harness on LangGraph: todo planning, virtual filesystem with permissions, subagents, SKILL.md skills with progressive disclosure, AGENTS.md memory
	Pydantic AI Harness [39]	"The batteries" for pydantic-ai: composable capabilities (filesystem, code mode in a Rust-sandboxed interpreter) that turn a framework agent into a coding agent
	Strands harness-sdk [40]	The merger in a repository name: a framework organization shipping "the agent harness itself" as an SDK

Both directions converge on the same artifact shape—loop + tools + skills + sub-agents + hooks + MCP + sessions—whether one starts from a terminal product or from an orchestration library. The convergence is itself validation: when LangChain, the framework whose absence from every harness runtime Observation 9 documents, finally built a harness, it independently adopted the

corpus's conventions (SKILL.md progressive disclosure, AGENTS.md memory, sub-agent spawning, todo planning). And the meta-harness of the next subsection completes the picture from above: its baseline installation imports `claude-agent-sdk` and `openai-agents` as production dependencies—the "frameworks" a 2026 orchestration layer builds on *are* the harness SDKs. The twin absence of 2025-era frameworks thus gets its historical resolution: harnesses did not adopt the frameworks; they replaced them, became importable themselves, and turned the question "which agentic framework should I use?" into "which harness do you already run?"

Интересно, что у обвязок реально видна часть платформенной экономики

14.3 Platform Economics: Marketplaces, Switching Costs, Governance

Интересно отдельно описать сетевые эффекты и понять в чем они

Three developments from the ninety-day window are textbook platform economics.

Marketplaces. Codex grew a plugin marketplace (GitHub/git/URL sources, in-conversation install approval, workspace sharing); skills registries appeared in four systems with trust tiers, provenance verification, and quarantine (Section 12.5); OpenClaw externalized its official plugins to npm. Capability distribution now has app-store mechanics—including the app store's security pathologies, as SkillProbe's audit of published skills shows [89].

Harness ⇔ Models
Harness ⇔ Users
Harness ↕ Cloud Providers

Switching costs. Codex ships a first-class *importer for Claude Code's on-disk state*: it detects `~/.claude/projects` session JSONL files, converts them into Codex rollout items under an import ledger, and offers to translate `~/.claude/settings.json` into `~/.codex/config.toml`. Vendors writing importers for each other's session stores is the stage of platform competition where user data becomes the moat—and the countervailing force is equally visible: OpenCode reads Claude Code's skills directory, OpenHands reads Claude Code's plugin manifests and hosts rival harnesses outright, and the meta-harness arbitrages every vendor's lock-in mechanisms into its adapter surface.

Enterprise governance. Codex's configuration stack now extends above the user—MDM-managed preferences, host-wide system files, and enterprise cloud-delivered bundles feed both base defaults and a constraint engine that can hard-restrict what user and project scopes may set. Google, meanwhile, announced Gemini CLI's transition toward an Antigravity-branded CLI—closed-source, where Gemini CLI is Apache-2.0 (Section 3.7)—and withdrew its consumer free tier, and the SpaceX–xAI merger plus the announced $60B Cursor acquisition (pending as of July 2026) priced the harness layer at acquisition scale. Governance layers, gated tiers, and acquisition-scale pricing are the signatures of platforms, not of standalone tools.

Интересная ставка от Databrics

14.4 The Meta-Harness Layer

Omnigent [25] is not a twelfth harness; it is a bet that the harness has become a commodity component and that the durable value sits one layer up. Databricks open-sourced it in June 2026 (Apache 2.0, ∼1M lines total, ∼312K of production Python): a four-tier process topology (server ⇄ host daemon ⇄ runner ⇄ per-conversation harness subprocess) whose adapter boundary is, by explicit design, a recursive subset of Omnigent's own public REST API. Its registry ships 23 canonical harness adapters (plus 16 aliases and a community entry-point group) spanning Claude Code, Codex, Cursor, OpenCode, Hermes, Pi, Goose, Qwen, Kimi, Kiro, Copilot, and Antigravity, formalized into five integration modes (`sdk-in-process`, `cli-subprocess`, `acp-subprocess`, `native-tui`, `native-server`)—and the declared capabilities are reconciled against a *conformance bench* (probes for basic turns, tool calling, streaming, interrupts, model override, policy denial), with live verification landed for the four flagship SDK adapters and best-effort declarations for the rest. Harnesses are tested like hardware.

Above that boundary the meta-layer adds four things no single harness provides. *Composition*: 1
any registry harness is addressable as a sub-agent session of any other, so a Claude-brained orchestrator can dispatch work to Codex and have Cursor review it. *Cross-harness policy*: one three-level 2

policy plane (session → agent → admin; six phases; CEL (Common Expression Language), Python, or LLM-classifier evaluators; cross-session per-user budgets) enforced on foreign harnesses *through each vendor's own extension mechanism*—Claude Code hooks, Cursor hooks, Hermes hooks, ACP permission requests—with fail-closed semantics. The lock-in mechanisms of Observation 2 become, from one layer up, an adapter surface. *A uniform sandbox and egress stack*: bubblewrap-plus-seccomp on Linux, generated Seatbelt profiles on macOS, Job Objects on Windows—the same architecture the corpus documents inside Codex, re-implemented at the meta-layer, plus an L7 MITM egress proxy (own CA, per-host rules, default-deny) hosting a *secretless credential proxy*, where real tokens never enter the sandbox and synthetic placeholders are swapped for credentials in flight—a design Codex also ships (its `network-proxy` crate), here lifted to the meta-layer across all wrapped harnesses. *Shareable sessions*: server-durable transcripts with multi-device sync, ACL grants, review comments, fork, and mid-session harness switching.

Just as informative is what it does *not* do: it implements no editing loop, no repository context, no edit-application strategy—the anatomy's D3/D4 core stays below the line—and it does not pretend the harnesses are equivalent: per-harness capability records, vendor-specific webhook endpoints, a Claude-specific `todos` field, and a Codex-only goal-mode extension leak through the "common" API by design. Its flagship example, Polly, is a cross-vendor coordinator-worker: a Claude Code-brained orchestrator that writes no code, fans work out to six vendor harnesses in per-task git worktrees, and mandates *cross-vendor review* in its orchestrator prompt—the reviewer must be a different vendor than the implementer (the mechanism-layer guardrails gate fan-out and blast radius, not vendor identity). Even its restraint is informative: sandboxing is applied inconsistently *by design* (it exec-wraps the Claude CLI in its own sandbox but *delegates* to Codex's native sandbox modes), evidence that OS-level isolation resists being factored out as a shared service. And the twin absences hold at the meta-layer: no agentic framework, no RAG, in a million lines—unless one counts its two baseline dependencies, `claude-agent-sdk` and `openai-agents`, which is precisely the point of Section 14.2.

Стало интересно попробовать этот meta harness

14.5 Ninety Days of Harness Evolution

Because the eight systems of the April edition were re-pinned rather than replaced, the corpus contains a controlled longitudinal sample: the same harnesses, source-diffed across one quarter. Four movements characterize the window.

Convergence became imitation. April's convergences were mostly independent rediscovery; July's are traceable. Codex adopted Claude Code's hook event vocabulary verbatim and its plan-mode ergonomics; OpenHands adopted Claude Code's plugin manifest format, its task-tool signature, and its static/dynamic cache boundary; OpenCode reads Claude Code's skills directory; Hermes's source comments credit OpenCode (edit matcher), Codex (smart approvals), OpenClaw (orchestrator prompt), and Goose (context hints). Cross-harness lineage is now written in the code itself.

Patterns diffused down the corpus. Deferred tool loading went from one system to three (plus seven skills variants); read-only plan modes from two to all four provider-native systems; LLM approval classifiers from one (Claude Code) to two (Codex's Guardian); turn-level checkpointing from one (Mistral Vibe) to three; safety-aware scheduler partitioning from one to two. The half-life of a competitive distinctive in this field is currently measurable in weeks.

Policy migrated out of prose. The clearest rhetorical trend of the quarter (Observation 3): Codex's newest model prompts dropped the no-commit and anti-gold-plating rules in favor of feature flags; Mistral Vibe deleted its "Never Commit" hard rule and re-founded its prompt on a seven-level precedence contract, A/B-tested server-side; OpenHands teaches commit mechanics. As models internalize norms and harnesses grow governance surfaces, behavioral policy is moving from the prompt (where the model reads it) to configuration (where the platform enforces it).

The trees themselves moved at platform speed. CODEX's workspace nearly doubled (621K → ∼1.12M lines of Rust, 89 → 126 crates) in one quarter—memories, Guardian, marketplace plugins, code mode, realtime voice; MISTRAL VIBE grew 77% (35.6K → 63K lines); OPENHANDS re-architected into an SDK while its application repo became an automation control center; GEMINI CLI announced its brand pivot and gated its free tier; AIDER, the field's pioneer, settled into community maintenance with 18 commits in the window. Three of the April edition's observations required substantive revision in place—the coupling dichotomy (Observation 2), the size-implies-sandbox correlation (Observation 6), and the protocol placement story (Observation 11)—each corrected by evidence that did not exist, or that we could not see, ninety days earlier. The methodological lesson generalizes: in this field, *inventory* claims (tool counts, feature cells, version pins) decay in weeks, while *structural* claims (loop taxonomy, subsystem anatomy, the absences) have so far proven durable. We have separated the two accordingly throughout this edition.

Observation 12. The coding-agent harness completed its platform turn in the first half of 2026, and every layer of the turn is now visible in source: extension substrates (hooks, skills, plugins) converged across all production systems; capability distribution acquired marketplaces, registries, trust tiers, and agent authors. Vendors shipped importers for each other's on-disk state and MDM-grade governance layers; the harnesses became importable SDKs while the framework vendors shipped harnesses (the merger of Section 14.2); the agent itself became addressable as a model behind an OpenAI-compatible endpoint; and a meta-harness now orchestrates a fleet of eleven vendor harnesses—five of the systems studied here among them—behind one API, re-implementing the expensive parts (sandbox, policy) and arbitraging the proprietary parts (hooks, session stores) of each. The competitive unit of the field is no longer the agent loop; it is the ecosystem surface around it.

Toxome, что это Tax и есть

15 Discussion

15.1 The Anthropic Effective-Agents Series as Empirical Validation

Between December 2024 and September 2025, Anthropic published four engineering articles offering prescriptive guidance on agent design: *Building Effective Agents* [16], *Effective Context Engineering for AI Agents* [19], *Writing Effective Tools for AI Agents* [18], and the case study *How We Built Our Multi-Agent Research System* [17]. Across the four pieces, a coherent design philosophy emerges: hand-rolled loops in preference to framework abstractions, a small set of high-signal tools in preference to wrap-everything APIs, just-in-time structural retrieval in preference to pre-indexed RAG, orchestrator-worker patterns in preference to flat monoliths, transparency about agent planning, and explicit context budgeting as an engineering discipline.

Eleven independently developed systems (only one of which, CLAUDE CODE, is from Anthropic) line up with these prescriptions:

- "Don't use frameworks unless necessary" [16]: 0 of 11 systems use LangChain, LangGraph, AutoGen, CrewAI, ADK, LlamaIndex, Pydantic AI, Genkit, or Semantic Kernel in the agent runtime (Observation 9).
- "ACI matters as much as HCI" [16] (the concept originates in this article; Aizawa et al. [18] extend it with practical tooling advice) and "few thoughtful tools" [18]: AIDER's 13 polymorphic edit formats, CLAUDE CODE's deferred tool loading, CODEX's custom patch format, HERMES's nine-strategy edit chain, and OPENCODE's model-conditional tool surfaces are all heavy ACI investments (Observation 4).
- "Just-in-time over RAG" [19]: 0 of 11 systems use vector embeddings over code; all use `grep`, tree-sitter, glob, and the file system, plus auto-discovered Markdown context files (Observation 9).

- "Orchestrator-workers for parallelizable, context-spanning tasks" [17]: emerges in Claude Code (recursive composition), Codex (thread tree with fan-out), Gemini CLI (registry + session protocol), Mistral Vibe (`task` tool), OpenHands (parallel delegation), Hermes (orchestrator role + swarm), and OpenCode (concurrent child sessions) (Observation 7).
- "Save plans to external memory before subagent spawn" [17]: Claude Code's CLAUDE.md, Gemini CLI's plan files and `MEMORY.md` project index, Codex's AGENTS.md hierarchy and its agent-maintained memories root.
- "Compaction + structured note-taking + sub-agents" for long-horizon tasks [19]: all four provider-native systems—and all three newcomers—implement all three.
- "Multi-agent systems use ∼15× more tokens than chat" [17] (vs. a single chat baseline, not a single-agent pipeline): this is what motivates Claude Code's prompt-cache-sharing fork mechanism, Hermes's cache-sharing background forks and summary-only returns, and Gemini CLI's thought-stripping (extended-thinking text is kept out of persisted history when recording responses, so it never re-enters the cache). Without those optimizations the cost would be prohibitive.

There is one tension in the guidance that we should call out. Hadfield et al. [17] caution that "most coding tasks involve fewer truly parallelizable tasks than research, and LLM agents are not yet great at coordinating and delegating to other agents in real time." Seven of the eleven systems in this study (eight counting OpenClaw's protocol-layer variant) nevertheless implement coordinator-worker patterns specifically for coding workflows. On closer inspection the observed patterns are mostly used for breadth-first exploration phases (parallel codebase research) rather than for parallel implementation, which is consistent with Hadfield et al.'s caveat once the unit of analysis shifts from the system as a whole to the individual workflow phase.

15.2 The Minimalism Argument

Mini-SWE-Agent's competitive self-reported benchmark performance (74%+ on SWE-Bench Verified) with a minimal scaffold and a single bash tool undercuts the assumption that rich tool ecosystems are necessary. The finding echoes the CodeAct [3] argument that executable code subsumes discrete tool actions. For benchmark evaluation alone, most of the scaffolding in the production systems turns out to be overhead.

The counter-argument is that benchmarks measure only a narrow slice of the value proposition. Production systems (Claude Code, Codex, Gemini CLI, Mistral Vibe, OpenHands) invest heavily in things that benchmarks do not reward:

- **Safety**: Preventing destructive actions on user codebases.
- **User experience**: Streaming responses, rich terminal UIs, progress tracking.
- **Extensibility**: Supporting custom tools, MCP servers, plugins.
- **Robustness**: Handling edge cases, stuck detection, error recovery.
- **Cost management**: Prompt caching, context compaction, model selection.

These concerns are invisible to SWE-Bench but critical for production adoption.

15.3 Safety as Architecture

Safety is the dimension on which the corpus is most spread out. Codex takes a defense-in-depth approach: OS-level sandboxing plus policy-as-code plus approval workflows, with safety treated as a primary engineering concern from the start rather than as a late addition. Gemini CLI sits at a similar but lighter point on the spectrum: a cross-platform sandbox (reusing OS binaries) plus four explicit approval modes (PLAN/DEFAULT/AUTO_EDIT/YOLO) and per-mode TOML policies.

Claude Code's three-layered system (hooks → classifier → dialog) sits between pure automation and human oversight. Mistral Vibe's permission-scope hierarchy (command/file/URL patterns plus agent-profile gates) is more granular than Aider's binary confirmation but has no OS-level enforcement. OpenHands's pluggable `SecurityAnalyzer` framework lets rule-based and LLM-based analysis be combined.

The April edition observed that the corpus's two largest systems (Codex, then at 621 K lines, and Gemini CLI at 568 K) were also its two cross-platform sandboxers, and read the correlation as structural. The expanded corpus falsifies the structural reading while preserving the cost claim: Hermes and OpenCode are comparably large with zero OS-level isolation, spending their safety budgets on content-borne threats and permission granularity respectively (Observation 6). What remains true is that a native sandbox, where built, is among the most code-expensive components in a harness—Codex *vendors bubblewrap into its tree*, and the meta-harness re-pays the entire bill one layer up (Section 14.4). The minimal safety surface in Aider and Mini-SWE-Agent is defensible for their target use case (developer tools with trusted users) but limits their applicability in enterprise or automated deployment contexts.

15.4 The Model–Agent Co-design Thesis

Пример совместной эволюции и так, чтобы можно было безрелизно обновлять

The coupling between agent scaffolds and foundation models runs in both directions.

The model shapes the agent. Codex's per-model prompts—now server-delivered catalog data spanning GPT-5.2 through the 5.6 family, with per-model tool modes and multi-agent tool generations—make explicit that scaffold design has to evolve with model capabilities, and that the vendor intends to drive that evolution without client releases. Claude Code's extended-thinking integration (budget tokens, thinking blocks) exploits Claude-specific reasoning features. Gemini CLI's `ModelRouterService` treats model selection itself as a runtime scheduling decision, dispatching each request to the cheapest sufficient Gemini variant—and absorbing a full model-generation shift (2.5 → 3.x defaults) within one quarter; the same idea is studied formally as Hybrid LLM [84], RouteLLM [83], and the FrugalGPT cascade [82]. Mistral Vibe's `reasoning_effort` mapping and `ThinkChunk` parsing exploit Mistral's reasoning enum. Aider's model-aware edit format selection adapts the editing strategy to each model's output patterns, and OpenCode re-derives the same idea in its tool registry (GPT-family models get a patch DSL; others get string replacement) and its nine-prompt model-family matrix. OpenHands ships model-specific tool *presets* (default, gemini, gpt5, planning)—co-design reaching even a multi-provider system's tool surface.

The agent shapes model usage. Claude Code's deferred tool loading reduces prompt tokens, changing the model's effective context window. Claude Code's prompt-caching boundary splits the system prompt to maximize cache hits, structuring the prompt around cache economics as much as around logical organization; Gim et al. [81] provide the systems-level grounding for the modular attention-reuse strategy this implicitly assumes. Gemini CLI keeps extended-thinking traces out of persisted history at response-recording time, so they never leak into future cache lookups (and strips thought signatures on auth switch). Mistral Vibe's middleware pipeline lets per-turn token-budget policies be composed orthogonally to the loop body. OpenHands's condensers selectively forget events, shaping what the model "remembers."

Implications. The four provider-native systems (Claude Code, Codex, Gemini CLI, Mistral Vibe) can optimize both sides of this coupling at once, evolving prompts, tools, and features in lockstep with model updates—Codex most literally, via its server-delivered model catalog. The April edition inferred that multi-provider systems must therefore design for the lowest common denominator; the expanded corpus shows that inference was too strong. Hermes, Pi, and OpenCode demonstrate that the full provider-native optimization menu is reachable from a multi-provider substrate by paying the per-provider conditional-code cost deliberately and centrally

(Observation 2)—per-model quirk metadata, provider-profile plugins, transform matrices. What the multi-provider systems cannot replicate is the *update loop*: only the vendor can re-tune scaffold behavior server-side on model-release day. MISTRAL VIBE's provider-first-with-generic-fallback design and the owned-transports position both remain viable middle paths, differing in who carries the maintenance burden.

не хватает визуализации этой кривой.

15.5 The Scaffold–Capability Frontier (a Guiding Intuition)

We close with a guiding intuition rather than a formal hypothesis: across a fixed task distribution and a fixed model, scaffold complexity and observed task success appear to trace a roughly concave curve. A floor below which the agent cannot operate at all (no loop, no tool, no memory, no result), a steep early-gain region where adding a small set of structural elements (a bash tool, a read/write tool, a Markdown context file) raises success rapidly, and a diminishing-returns plateau where further scaffold work pays off in operational concerns (safety, UX, extensibility) rather than in completion rate. We label this picture the *scaffold–capability frontier* and treat it as a hypothesis to be tested, not as a proven property of the design space.

Two independent results in the literature are consistent with this picture without confirming it. MINI-SWE-AGENT's reported ∼74%+ on SWE-Bench Verified with a deliberately minimal scaffold suggests that, for SWE-Bench-style tasks on a frontier model, the floor is already well below most production scaffolds. Lin et al. [85] report that their *Agentic Harness Engineering* (AHE) system, starting from a bash-only seed comparable to MINI-SWE-AGENT and automatically evolving the harness using observability-driven feedback, reaches 71.9% on SWE-Bench Verified. Their component-level ablation finds that *structural* harness elements carry the improvement (tools +3.3 pp, middleware +2.2 pp, long-term memory +5.6 pp), while the system prompt alone *regresses* performance (−2.3 pp); the evolved harness also transfers across four model families. We read this as suggestive evidence that structural scaffold is more portable than prose-level prompt strategy, consistent with our model–agent co-design discussion at the architectural (rather than weights) level.

What an operational definition would require is at minimum: a fixed task distribution T, a fixed model M, a metric for scaffold complexity (LoC, tool count, structural feature count, or some weighted combination), and a target success rate p. A "minimum viable scaffold for (T, M, p)" would then be the lower envelope of scaffold-complexity values whose empirical success rate on T with M exceeds p. We do not produce such a measurement here. Doing so would require a controlled cross-system run study that this paper deliberately avoids (see Threats to Validity, Section 15). We flag the scaffold–capability frontier as a question for follow-on empirical work, and note that the relevant minimum almost certainly varies across:

- **Task complexity**: Multi-file refactoring may require more sophisticated tool support than single-file bug fixes.
- **Safety requirements**: Enterprise deployments demand scaffolding that benchmarks don't measure.
- **Interaction patterns**: Long-running interactive sessions benefit from memory management that short benchmark runs don't need.

15.6 Threats to Validity

A study of this shape has real limits, and it is worth being explicit about them.

The analysis rests on source-code reading, not runtime measurement. We do not claim to know how fast any of these systems *are*; we claim to know how they are *structured*. Where the paper reports benchmark numbers, they come from the systems' own documentation, and the last decimal place is not the interesting part.

The qualitative scoring underlying the architectural comparisons involves judgment calls. We tried to anchor each score in a specific implementation detail, and we welcome disagreement from

anyone who reads the same codebases and scores them differently. We deliberately avoid line-number citations, and we date all code-size figures to the July-2026 pins: every system in the corpus is actively developed, and finer precision would be obsolete by the time the paper is read. All versions are pinned to exact release tags and commits (Table 3), the April-2026 snapshots are retained for the longitudinal comparison of Section 14.5, and—a lesson that quarter taught us directly—we now distinguish *inventory* claims, which we date, from *structural* claims, which we expect to endure. Three observations of the April edition were substantively revised in this one on new evidence; the revisions are integrated in place and flagged where they occur.

The Claude Code analysis is the weakest link on reproducibility. It draws on a publicly circulated source snapshot from March 2026 rather than an official release, and the shipping binary has evolved considerably since (we track its changelog-level evolution but do not treat changelog claims as source evidence). The architectural claims should still be testable against the public `claude-agent-sdk` and the behavior of the shipping binary, but we acknowledge the asymmetry with the other ten systems, whose source trees are trivially re-derivable from public Git history.

The framework-absence finding (Observation 9) is strong but structurally conservative. We checked dependency manifests and performed import grep across three languages—twice, three months apart, across twelve trees including the meta-harness; we did not trace internal forks, plugins loaded via dynamic `importlib`/`require`, or transpiled distributions. A company could in principle use LangChain inside a production SWE agent through a code path we did not inspect. We would be surprised, but not shocked.

The Anthropic-guidance-to-observation mapping (Observation 10) is suggestive but does not by itself establish causation. The right follow-up is interview work with engineers at each vendor; we did not do that.

We also compared each system independently rather than running all eleven on a shared task set. A head-to-head execution study would turn some of the qualitative scoring into measurable quantities. It would also cost an order of magnitude more effort than this paper, and we chose the tractable scope.

16 Design Recommendations: A Practitioner's Guide

The preceding sections are descriptive, cataloging what the eleven production harnesses actually contain. This section is prescriptive: we distill the observations into actionable recommendations for an engineer designing a new coding agent. Each recommendation cites the observations that support it, names the systems that implement it, and identifies the trade-off that might justify a different choice. Where the corpus agrees with the Anthropic *Effective Agents* series [16–19], we flag the agreement as cross-source validation. Where the corpus diverges from or extends the series, we note that too. The recommendations are grouped by subsystem and ordered roughly in the sequence a practitioner would encounter them when bootstrapping an agent.

16.1 Loop Architecture

Recommendation 1. Start with a linear `while` loop; graduate to a middleware pipeline only when orthogonal turn policies emerge. *Evidence*: Observation 1 (loop sophistication does not predict benchmark performance); Mini-SWE-Agent's 50-line linear loop reports 74%+ on SWE-Bench Verified (self-reported; see the footnote to Table 4). Upgrade path: once you need three or more independent turn-level policies (turn limits, cost caps, auto-compaction, context-budget warnings, read-only mode), adopt Mistral Vibe's middleware pipeline pattern—each policy becomes a composable middleware, not a new branch in the loop body.

16.2 Provider Coupling

Recommendation 2. If you ship a foundation model, couple tightly to your home provider and expose a generic fallback adapter for portability; if you don't, provider-native optimizations are still on the table—budget for per-model metadata instead. *Evidence*: Observation 2. MISTRAL VIBE's "provider-first with generic fallback" remains the vendor pattern. For multi-provider harnesses, PI's per-model `compat` quirk flags, HERMES's declarative provider profiles, and OPENCODE's transform matrix show that cache breakpoints, thinking tiers, and reasoning effort are reachable from an abstraction—the cost is a maintained per-provider conditioning layer, paid once and centrally.

16.3 Tool Design

Recommendation 3. Begin with just a `bash` tool. Add more tools only in response to observed failure modes. *Evidence*: MINI-SWE-AGENT's single-tool design reports 74%+ on SWE-Bench with essentially no tool infrastructure (self-reported); Aizawa et al. [18] caution that "more tools don't always lead to better outcomes." A common sequence is: add `read_file` and `write_file` when bash output truncation becomes a problem; add `grep/glob` when bash+`find/rg` feels awkward; add `search_replace` when full-file writes waste tokens.

Recommendation 4. When your tool count exceeds ∼15, adopt deferred tool loading. *Evidence*: CLAUDE CODE's `shouldDefer` flag + `ToolSearchTool` reduces the initial prompt by ∼40%; CODEX independently converged with `defer_loading` flags plus a BM25-ranked `tool_search`; HERMES collapses overflowing MCP catalogs into three BM25-searched bridge tools once schemas would exceed 10% of the context window. Below ∼15 tools the indirection is not worth the added complexity; above it, the prompt bloat becomes prohibitive.

16.4 File Editing

Recommendation 5. Match your edit-tool contract to your model tier: exact unique-substring replacement for frontier models, a fuzzy cascade for open or weaker models—and in either case handle drift *at the tool*, never by line numbers. *Evidence*: Observation 4. The frontier-model camp has converged on exact contracts (CLAUDE CODE; MISTRAL VIBE deleted its SEARCH/REPLACE tool and converged on exact matching mid-2026; PI adds only Unicode/whitespace canonicalization with byte-fidelity overlay). The drift-tolerant camp serves broader model ranges (OPENCODE's nine-stage cascade at Levenshtein 0.65, HERMES's nine-strategy chain, AIDER's `RelativeIndenter`). GEMINI CLI's LLM edit-fixer subcall is the emerging third option: repair the match with a model instead of a threshold. Avoid line-number-based edits: models drift on line numbers more than on contextual matching.

16.5 Memory and Context

Recommendation 6. Auto-discover hierarchical Markdown context files at project, user, and extension scopes—and read your neighbors' filenames too. *Evidence*: all eleven systems that manage repo context converged on the pattern, and the newest ones read *multiple* conventions (HERMES: own file, then AGENTS.md, CLAUDE.md, .cursorrules; OPENCODE: AGENTS.md/CLAUDE.md/CONTEXT.md plus remote URLs; OPENHANDS ingests three ecosystems' files as scoped rules). Inject top-level content near the top of the system prompt, surface *nested* files just-in-time when tools touch their subtree (MISTRAL VIBE, OPENCODE, HERMES, GEMINI CLI), and let the model persist durable facts—by direct edits, bounded snapshot files, or a human-reviewed inbox (Observation 5).

Recommendation 7. Implement threshold compaction at a fixed buffer below the model's context window; preserve a verbatim recent tail; merge summaries incrementally rather than re-summarizing; and wire the same routine to fire reactively on overflow. *Evidence*: Claude Code fires below a 13 K-token buffer and post-restores files; Gemini CLI compacts at 50% preserving the last 30%; Pi and OpenCode pass the *previous* summary back for merging (anchored/iterative summaries), which preserves early decisions that from-scratch re-summarization loses; Mistral Vibe re-injects prior user messages into the compacted envelope and reuses the routine on mid-turn `ContextTooLong` errors; OpenHands routes overflow errors into condensation. Aggressive compaction protects against context rot [19]; conservative preservation keeps recent reasoning coherent.

Recommendation 8. Do *not* build RAG over code. Use `ripgrep`, `glob`, tree-sitter symbol extraction, and file-system traversal instead. *Evidence*: Observation 9; 0/11 systems use vector embeddings for code retrieval; Rajasekaran et al. [19] favor JIT retrieval in practice, and all their concrete recommendations point toward deterministic tools. Code has rich deterministic structure (paths, language servers, tree-sitter parses) that semantic similarity cannot replicate, and changes minute-to-minute, making embeddings stale.

16.6 Safety Architecture by Deployment Context

Recommendation 9. For a **developer tool (semi-trusted)** context: implement a three-mode approval system (PLAN / DEFAULT / YOLO) with permission-scope patterns. *Evidence*: Gemini CLI's PLAN/DEFAULT/YOLO modes, Mistral Vibe's 4-layer permission hierarchy (tool-wide + tool-specific + session rules + interactive callback); matches how a human developer triages risk.

Recommendation 10. For **enterprise / shared / automated** contexts: implement OS-level sandboxing with policy-as-code and per-agent audit trails. *Evidence*: Observation 6; Codex and Gemini CLI ship native cross-platform sandboxes (Bubblewrap on Linux—vendored in Codex's case—Seatbelt on macOS, restricted tokens on Windows), and Claude Code wraps Anthropic's reusable `sandbox-runtime` as an opt-in. Gemini CLI's implementation shows the cost is tolerable when you reuse OS binaries (Node `child_process` or equivalent) rather than writing namespace plumbing from scratch.

Recommendation 11. Regardless of tier, codify safety rules as data or dedicated policy files rather than as imperative code—and if you support a YOLO mode, keep a floor beneath it. *Evidence*: Codex's Starlark `execpolicy` rules with parse-time-validated inline examples, Claude Code's `PreToolUse` hooks, Gemini CLI's per-mode TOML, OpenCode's last-match-wins rulesets. Hermes adds the floor lesson: its twelve hardline patterns survive `-yolo`, with the bypass flag frozen at module import so injected content cannot flip it at runtime. Policies-as-code survive refactors and are auditable independently of the loop implementation.

16.7 Multi-Agent Orchestration

Recommendation 12. Stay single-agent until you can point to a concrete breadth-first exploration phase where parallel context isolation clearly beats serial search. *Evidence*: Hadfield et al. [17] caution that multi-agent systems use ∼15× more tokens than a chat baseline and that "most coding tasks involve fewer truly parallelizable tasks than research"; Mistral Vibe's `task` tool delegates to sub-agents (its prompt encourages launching several in parallel), which is sufficient for most coding workflows.

Recommendation 13. Ship an ACP *server*: it is no longer just an editor integration—it makes your harness consumable by hosts and meta-orchestrators. Keep your own sub-agents in-process. *Evidence*: Observation 11. An ACP server now buys three audiences at once: IDEs (Zed, JetBrains), hosting harnesses (OpenHands runs Claude Code/Codex/Gemini CLI as interchangeable ACP backends), and meta-harnesses (Omnigent drives Goose and Qwen over ACP). Six of eleven corpus systems ship it. For **agent** ↔ **agent** mesh topologies, **A2A** remains Gemini CLI's bet alone—defensible, unproven. For **main-agent** ↔ **sub-agent** coordination within your own runtime, do *not* adopt either (see Recommendation 12): in-process primitives are what production systems use, and even the cross-process exceptions (Pi's JSONL extension, Hermes's SQLite-blackboard swarm) avoid the standard protocols for that role.

16.8 Extensibility

Recommendation 14. Use **Skills** for capability templates (workflows, domain knowledge, procedural recipes) and **MCP** for external integrations (Slack, databases, internal APIs)—in that order of priority. *Evidence*: Observation 8: skills now lead MCP in adoption (9/11 vs. 8/11), discovery is cross-vendor (OpenCode reads `~/.claude/skills`), and distribution has registries and provenance. MCP remains the right layer for running external processes; Pi demonstrates the defensible minimal position (skills plus CLI tools with READMEs, no MCP). If you install third-party skills, treat them as packages: trust tiers, scanning, and quarantine (Hermes) are the emerging baseline [89].

16.9 What Not to Build (Anti-Patterns)

Recommendation 15. **Do not** use LangChain, LangGraph, AutoGen, CrewAI, LlamaIndex, Pydantic AI, Genkit, Google ADK, or Semantic Kernel for the agent runtime. *Evidence*: Observation 9; 0/11 production harnesses in the corpus use these frameworks—Gemini CLI skips Google's own. Schluntz & Zhang [16] explicitly warn: "frameworks often create extra layers of abstraction that can obscure the underlying prompts and responses, making them harder to debug." Use raw SDK calls—or, the 2026 corollary (Section 14.2): if you want a batteries-included starting point, the harness SDKs (Claude Agent SDK, `openai-codex`, `openhands-sdk`) *are* the framework layer now, with the corpus's patterns built in.

Recommendation 16. **Do not** build a vector-embedding retrieval layer for code. *Evidence*: Observation 9; 0/11 systems do this. The embeddings that do appear serve *conversation memory* (OpenClaw's default hybrid search; Hermes's opt-in plugins), and Hermes shows lexical SQLite FTS5 sufficing for that role at 642K-line scale. If you believe your domain needs semantic code retrieval, first prove it improves over `ripgrep`+`tree-sitter` on a held-out task set before adding the infrastructure.

Recommendation 17. **Do not** wrap every upstream SaaS API as a 1-to-1 tool. *Evidence*: Aizawa et al. [18]; "a common error we've observed is tools that merely wrap existing software functionality or API endpoints." Consolidate instead.

Recommendation 18. **Do not** over-engineer stuck detection—but do ship the cheap caps, which cost a dozen lines. *Evidence*: Claude Code and Codex still ship no automated stuck detection and deliver production-quality agents. The floor moved, though: even Mini-SWE-Agent now caps consecutive malformed responses and wall-clock time, OpenCode's doom-loop check is a three-identical-calls counter routed to a permission ask, and Hermes hashes call signatures but ships the hard stop *disabled*, trusting warnings. OpenHands's five-scenario `StuckDetector` and Gemini CLI's hybrid hash+LLM service remain platform-tier investments.

16.10 A Minimum Viable Harness

As a concrete starting point, Listing 3 sketches a minimum viable harness in ∼90 lines of Python, combining the patterns recommended above: linear loop (Mini-SWE-Agent), middleware-style policies (Mistral Vibe), four-tool surface (bash, read, write, search_replace), hierarchical Markdown context auto-discovery (all four provider-native systems), and threshold compaction (Claude Code/Gemini CLI/Mistral Vibe). It is not a drop-in library; it is a scaffold to be copied and specialized.

```python
from __future__ import annotations
import asyncio, json, pathlib, subprocess
from dataclasses import dataclass, field
from typing import Any, Protocol

# ---- Provider abstraction (Recommendation 2): provider-first fallback ---------
class Model(Protocol):
    async def complete(self, messages: list[dict], tools: list[dict]) -> dict: ...

# ---- Tools (Recommendation 3): bash + 3 file tools --------------------------
def _truncate(s: str, n: int = 25_000) -> str: return s if len(s) <= n else s[:n] + "\n...[truncated]"

def tool_bash(cmd: str) -> str:
    r = subprocess.run(cmd, shell=True, capture_output=True, text=True, timeout=120)
    return _truncate(f"exit={r.returncode}\nstdout:\n{r.stdout}\nstderr:\n{r.stderr}")

def tool_read_file(path: str, offset: int = 0, limit: int = 2000) -> str:
    lines = pathlib.Path(path).read_text().splitlines()
    return _truncate("\n".join(f"{i+1:4}: {ln}" for i, ln in enumerate(lines[offset:offset+limit])))

def tool_write_file(path: str, content: str) -> str:
    pathlib.Path(path).write_text(content); return f"wrote {len(content)} bytes"

def tool_search_replace(path: str, search: str, replace: str) -> str:
    p = pathlib.Path(path); text = p.read_text()
    if text.count(search) != 1: return f"ERROR: search string occurs {text.count(search)}x; must be unique"
    p.write_text(text.replace(search, replace, 1)); return "OK"

TOOLS = {"bash": tool_bash, "read_file": tool_read_file,
         "write_file": tool_write_file, "search_replace": tool_search_replace}

# ---- Hierarchical Markdown context (Recommendation 6) ----------------------
def discover_context(cwd: pathlib.Path = pathlib.Path.cwd()) -> str:
    parts = []
    for p in reversed([cwd, *cwd.parents]):                    # root-to-leaf
        md = p / "AGENTS.md"
        if md.exists(): parts.append(f"<ctx path='{md}'>\n{md.read_text()}\n</ctx>")
    return "\n".join(parts)

# ---- Middleware pipeline (Recommendation 1) --------------------------------
@dataclass
class Agent:
    model: Model
    messages: list = field(default_factory=list)
    n_turns: int = 0
    cost: float = 0.0
    max_turns: int = 50
    max_cost: float = 5.00
    compact_at_tokens: int = 120_000

    def _token_estimate(self) -> int:
        return sum(len(json.dumps(m)) for m in self.messages) // 4

    async def _check_limits(self):
        if self.n_turns >= self.max_turns: raise StopIteration(f"max_turns={self.max_turns}")
        if self.cost   >= self.max_cost:   raise StopIteration(f"max_cost=${self.max_cost:.2f}")

    async def _maybe_compact(self):                            # Recommendation 7
        if self._token_estimate() < self.compact_at_tokens: return
```

```
        summary = await self.model.complete(
            self.messages + [{"role": "user",
                "content": "Summarize the conversation. Preserve decisions and unresolved issues."}], tools=[])
        # Preserve system + last 30% of turns verbatim, drop the middle (Gemini pattern)
        keep = max(4, int(len(self.messages) * 0.30))
        self.messages = [self.messages[0], {"role":"assistant","content":summary["content"]}] + self.messages[-keep:]

    async def run(self, task: str) -> str:
        self.messages = [
            {"role": "system", "content": f"You are a SWE agent.\n\n{discover_context()}"},
            {"role": "user",   "content": task},
        ]
        tool_schemas = [{"name": n, "description": f.__doc__ or n} for n, f in TOOLS.items()]
        while True:
            await self._check_limits(); await self._maybe_compact()
            resp = await self.model.complete(self.messages, tools=tool_schemas)
            self.n_turns += 1; self.cost += resp.get("cost", 0.0)
            self.messages.append(resp)
            if not resp.get("tool_calls"): return resp.get("content", "")
            for call in resp["tool_calls"]:                          # serial execution
                try:    out = TOOLS[call["name"]](**call["args"])
                except Exception as e: out = f"ERROR: {type(e).__name__}: {e}"
                self.messages.append({"role": "tool", "tool_call_id": call["id"], "content": _truncate(str(out))})
```

Listing 3: A minimum viable harness, ∼90 LoC Python. Illustrative scaffold, not production code.

The scaffold above deliberately omits features for which our corpus shows divergence: sandbox (Recommendations 9–10 are deployment-specific), multi-agent (Recommendation 12 defers it), MCP/Skills (Recommendation 14 is extensibility, not core). Start here, measure [17], add the minimum that your observed failure modes demand.

Observation 13. The 90-line minimum viable agent in Listing 3 implements 10 of the 18 recommendations directly and is compatible with the remaining eight. It does so with no framework dependencies, no RAG, no vector store, no multi-agent orchestration, and no sandbox, which lines up with the twin absences (Observation 9) and the "start simple" principle of [16]. We conjecture, without proof, that this scaffold running on a frontier model would match Mini-SWE-Agent's SWE-Bench numbers, and that moving beyond those numbers is primarily a model-capability question (Observation 1) rather than a scaffold question.

17 Conclusion and Future Work

This paper has laid out a source-code-level anatomy of eleven LLM-powered coding harnesses—one per major commercial LLM provider (Claude Code/Anthropic, Codex/OpenAI, Gemini CLI/Google, Mistral Vibe/Mistral) plus seven open-source designs—together with a meta-harness contrast point, a longitudinal diff of one quarter's evolution, and an account of the field's turn into platforms.

17.1 Key Takeaways

1. **Architecture matters, but not in the way the current discourse implies.** The sophistication of the agent loop does not predict benchmark performance (Mini-SWE-Agent's 50-line loop reports results in the frontier range). It does, however, predict production readiness: safety infrastructure, user experience, extensibility—and increasingly clients and transport, where the largest systems now carry most of their mass.

2. **The model–agent relationship is co-evolutionary, and coupling is about the update loop, not capability.** Provider-native optimizations (cache boundaries, thinking budgets, model-specific prompts, routing) are exercised from *both* ends of the coupling spectrum: the

vendor systems natively, and HERMES, PI, and OPENCODE from multi-provider substrates by paying the per-provider conditioning cost centrally. What only vendors retain is server-side co-evolution—CODEX re-tunes prompts, reasoning tiers, and even its multi-agent tooling per model release from a catalog endpoint, no client update required.

3. **Safety is architecturally expensive—and a choice, not a consequence of scale.** CODEX's cross-platform sandboxing (now with vendored bubblewrap) remains a substantial investment benchmarks never measure, and the meta-harness re-pays the same bill one layer up. But the April size-implies-sandbox correlation broke: HERMES and OPENCODE are among the largest systems with zero OS-level isolation, spending instead on content-borne threat defense and syntax-aware permissioning.

4. **Multi-agent orchestration converged on hierarchical patterns, and the protocols found a third role.** Coordinator-worker shapes now appear in seven systems. ACP serves not only the editor ↔ agent boundary (six corpus adopters) but a role outside its design brief: *harness hosting*, with OPENHANDS running CLAUDE CODE, CODEX, or GEMINI CLI as interchangeable backends. A2A remains GEMINI CLI's cross-vendor mesh bet. Sub-agent coordination stays in-process in eight of the nine multi-agent systems.

5. **The extensibility standards resolved: skills lead.** Skills (9/11 systems) overtook MCP (8/11) when PI broke the tie—implementing agentskills.io while rejecting MCP outright. The skills layer acquired registries, trust tiers, provenance verification, cross-vendor discovery (OPENCODE reads `~/.claude/skills`), and its first agent authors (HERMES's self-improving loop, GEMINI CLI's extraction inbox).

6. **The twin absences are real, re-verified, and now historically explained.** Across eleven systems and roughly four million lines, no agent runtime uses a general-purpose agentic framework, and none uses RAG over code—all rely on hand-rolled async loops and deterministic structural retrieval. The absence held through a threefold corpus expansion and a three-month re-audit, corroborating Anthropic's published guidance [16, 19] and the independent AHE result [85]. Its resolution is the harness–framework merger of Section 14.2.

7. **The Anthropic effective-agents series anticipates the architectures we observe.** The four engineering articles published between December 2024 and September 2025 map closely onto the patterns documented across eleven independently developed systems. Whether this reflects shared empirical reality, guidance-shaped design, or both is a question worth pursuing on its own.

8. **The analysis is directly actionable.** Section 16 distills 18 concrete, evidence-anchored design recommendations for practitioners building new harnesses, together with a 90-line minimum-viable-harness scaffold (Listing 3) that implements ten of them directly. Combined with the source-code audit, this is, as far as we know, the first evidence-based how-to for the discipline that is not a single-vendor whitepaper.

9. **The platform turn is no longer a hypothesis.** What the April edition advanced cautiously as the CLI-as-framework hypothesis, Section 14 now documents as an accomplished turn: the harnesses ship as importable SDKs while the framework vendors ship harnesses (the merger of Section 14.2); capability distribution has marketplaces, registries, trust tiers [89], and agent authors; vendors write importers for each other's on-disk state and expose MDM-grade (mobile-device-management) governance; the agent itself is addressable as a model behind an OpenAI-compatible endpoint; and a meta-harness orchestrates eleven vendor harnesses—five of the systems studied here among them—behind one API (Section 14.4). The surrounding ecosystem data points the same way: MCP's 8 000+ server ecosystem with app-store curation pathologies [86], skills formalized as composable packages [87] with npm-like package managers [88], 22–29% of GitHub projects bearing agent traces [91], and the "SE 3.0" framing of development as orchestration [92]. The developer no longer programs *with* the framework but

inside it, and the vendor lock-in of Observation 2 is becoming the structural lock-in that operating systems and IDEs have always produced [90]—with the meta-layer already arbitraging it. Whether harnesses consolidate into a few dominant platforms, fragment into interoperable runtimes, or commoditize from above is the competitive question of the next year; the architectural preconditions for all three outcomes are in the source code today.

17.2 Future Work

Several directions emerge from our analysis:

Интересный набор. Можно что-нибудь из этого забрать себе в исследованиях

- **Unified evaluation frameworks** that assess safety, user experience, cost efficiency, and extensibility alongside correctness, addressing the gap between benchmark performance and production readiness.
- **Reference architecture specification** that formalizes the patterns we identified (event sourcing, policy-as-code, context forking, deferred tool loading, middleware pipelines, hash-based loop detection, JIT repo context) into a reusable architectural framework.
- **Formal verification of safety policies** for policy-as-code systems, establishing guarantees about what agent actions are permitted under given policy configurations.
- **Empirical study of model–agent co-evolution**, tracking how agent scaffolds change across model generations and quantifying the performance impact of model-specific optimizations. Lin et al.'s AHE system [85] demonstrates that automatic harness evolution is already feasible; longitudinal studies could now track how automatically evolved scaffolds compare to hand-crafted ones across model generations.
- **Cross-system benchmark suite** that runs identical tasks across all eleven systems under controlled conditions, enabling direct comparison of the architectural trade-offs we identified.
- **Inter-agent protocol adoption study**, tracking ACP's three roles (editor integration, harness hosting, and—via A2A—the cross-vendor mesh) and measuring the overhead of hosted-harness topologies such as OpenHands's ACP backends and Omnigent's adapter fleet.
- **Longitudinal continuation**: the April/July snapshot pair turns this study into a time series; quarterly re-pinning would measure pattern-diffusion half-lives, prompt thinning, and the platformization trajectory quantitatively rather than anecdotally.
- **Meta-harness economics**: whether commoditization-from-above (Section 14.4) captures value durably, and whether the conformance-bench approach to harness capabilities becomes a standard interface.
- **Causation analysis of the Anthropic–industry alignment** via interviews, design-doc archaeology, and commit-history correlation, distinguishing public-guidance influence from convergent engineering.

Acknowledgments and Disclosure of AI Assistance

This paper was written with substantial assistance from Anthropic's Claude (via the Claude Code CLI), which was used both for the source-code analysis and for drafting the manuscript. The architectural findings, observations, patterns, and recommendations were verified against the referenced codebases before being kept in the paper. Any remaining errors are ours.

This disclosure follows the emerging authorship policies of ACL, NeurIPS, ICML, and the IEEE. The value of the paper, if it has any, lies in what it says about the twelve codebases; the named modules, tools, and version pins cited throughout should let any reader verify the claims independently.

Теперь в научных статьях надо это указывать.

We thank the maintainers and contributors of the twelve studied systems for working in the open. This paper would not have been possible without their decision to make their source code available for inspection.

A Detailed Comparison Tables

This appendix collects the densest per-system comparison tables, referenced from the body: the rhetorical content of the eleven system prompts (Table 16), the advanced API-feature matrix (Table 17), and the head-to-head Claude Code-versus-Codex pipeline comparison (Table 18).

Table 16: Rhetorical content of the eleven system prompts (July 2026).

System	**Verbosity rule**	**Anti-gold-plate**	**Commit policy**	**Emoji rule**	**Emphasis style**
Claude Code	quantified ($\leq$ 25/100 w; internal A/B)	explicit	no-commit unless asked	conditional	`IMPORTANT:`/`NEVER` tags
Codex	qualitative (5.2); persona-driven (5.6)	explicit (5.2); dropped (5.6)	no-commit (5.2); feature-flag (5.6)	none	`NEVER` tags
Gemini CLI	via snippet gating	implicit	implicit	none	feature-gated sections
Mistral Vibe	strict (<150 w)	explicit	*reversed*: commits expected, signature trailer	strict ban	overridability contract (structural)
OpenHands	none	explicit (file-suffix)	commits taught; push/PR gated	none	XML role tags
Aider	"few short sentences"	implicit	silent	silent	format examples
Mini-SWE-Agent	structural (1 cmd/turn)	silent	silent	silent	`<important>` XML
Hermes	qualitative ("lead with the change")	explicit	no-commit/push/rewrite unless asked	none	`MUST`/`NEVER` + exemplar pairs; model-gated XML
Pi	single bullet ("Be concise")	silent (delegated)	silent (delegated)	silent	XML data tags only
OpenCode	quantified (<4 lines; model-dep.)	explicit	no-commit unless asked	conditional; GPT prompt bans	`IMPORTANT:`/`NEVER` + `<system-reminder>`
OpenClaw	per-agent	inherited	inherited	inherited	per-agent

Table 17: Advanced API features used by each system (July 2026).

System	Prompt caching	Ext. thinking	Reasoning effort	Model routing	WebSocket	Vision	Cost tracking
OpenHands	Partial (cache tiers + markers)	✓	✓	✓(RouterLLM)	✗	✓	Per-model
Aider	Partial	✗	✗	✗	✗	Opt.	Tok+$
Claude Code	Full (Blake2b boundary)	✓	✗	✗	✗	✓	Tok+$
Codex	Partial (`thread_id` key)	✗	✓(to max/ultra)	✗	✓	✓	Per-turn
Gemini CLI	✗(server-side)	✓	✗	✓(router)	✗	✓	OTel/Tok+$
Mistral Vibe	Partial	✓	✓(5 levels)	✗	✗	✓	OTel/Per-sess.
Mini-SWE-Agent	Partial	✗	✗	✗	✗	Opt.	litellm
Hermes	Full (provider-agnostic markers + prefix normalization)	✓	✓(5 levels, per-provider translation)	Failover chains only	✗	✓	Tok+$, typed provenance
Pi	Full (breakpoints + TTL tiers + waste audit)	✓(7-level scale)	✓(10 dialects)	✗(by design)	✓(Codex back-end)	✓	Tok+$, tiered pricing
OpenCode	Full (6-dialect fanout + session key)	✓	✓(date-gated variants)	Small-model selection only	Exp. flag	✓	Tok+$ (Decimal)
OpenClaw	Partial	✓	✗	✗	✓	✓	Per-prov.

Table 18: Head-to-head pipeline comparison: Claude Code vs. Codex.

Stage	Claude Code	Codex
Prompt assembly	12–15 named sections. Static prefix (identity, rules, tools) cached via Blake2b hash; dynamic suffix (env, memory, MCP) recomputed per-turn. `SYSTEM_PROMPT_DYNAMIC_BOUNDARY` separates the two.	Per-model prompts as server-delivered catalog data (`models.json` `base_instructions`, remote-refreshed with bundled fallback); personality-templated. AGENTS.md injected via root-to-cwd concatenation (nested content appended last).
Tool exposure	43 tools, but only a subset loaded initially. Deferred tools discovered on-demand via `ToolSearchTool` (keyword search + `select:` syntax). Reduces prompt tokens by ∼40%.	25–30 built-ins; schemas converted to OpenAI `Function` format via `create_tools_json_for_responses_api()`. Deferred loading now mirrored: `defer_loading` flags (MCP tools by default) + BM25 `tool_search`.
API protocol	Anthropic Messages API. SSE streaming with `content_block_{start,delta,stop}` events. Extended thinking via `budget_tokens`.	OpenAI Responses API. WebSocket primary, SSE fallback. Connection prewarm (`generate=false`). Turn state via `x-codex-turn-state` header. Reasoning effort (minimal–xhigh, plus max/ultra; `ultra` adds automatic delegation).
Tool dispatch	`partitionToolCalls()`: single-pass reduce partitions by `isConcurrencySafe()`. Consecutive safe tools batched; unsafe tools solo. Max 10 concurrent via semaphore.	`FuturesOrdered` through a dedicated `ToolCallRuntime`; per-tool `supports_parallel` flag (default false) gates an execution lock—concurrency-unsafe tools run exclusively.
Permission	3 layers: (1) `PreToolUse` hooks (static pattern match), (2) LLM risk classifier, (3) interactive dialog. Sub-agents skip layer 3.	4 layers: (1) ExecPolicy Starlark rules, (2) lifecycle hooks (Claude Code-compatible vocabulary), (3) Guardian LLM approval reviewer (fail-closed), (4) OS-level sandbox enforcement.
Execution	Host execution with an opt-in OS sandbox (Anthropic `sandbox-runtime`: Bubblewrap/Seatbelt, network restrictions, `dangerouslyDisableSandbox` escape hatch); git worktrees for branch isolation.	Sandboxed execution. Vendored Bubblewrap (Linux; Landlock legacy fallback), Seatbelt (macOS), restricted tokens (Windows).
Compaction	Triggered when remaining context drops below `AUTOCOMPACT_BUFFER_TOKENS` (13 K), with an env-overridable percent threshold. Image stripping, message grouping by API round, LLM summarization, `SystemCompactBoundaryMessage`. Post-cleanup: restore 5 files (50 K token budget).	`model_auto_compact_token_limit` configurable. Local or cloud compaction via `/responses/compact`; rollout token budgets with turn aborts.

References

[1] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *Proc. ICLR*, 2024.

[2] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Proc. NeurIPS*, 2024.

[3] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. Executable code actions elicit better LLM agents. In *Proc. ICML*, 2024.

[4] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *Proc. ICLR*, 2023.

[5] L. Weng. LLM-powered autonomous agents. *Lil'Log*, 2023. https://lilianweng.github.io/posts/2023-06-23-agent/.

[6] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 2024.

[7] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proc. ICLR*, 2024.

[8] Q. Wu, G. Banber, Y. Zhang, Y. Wu, B. Li, Z. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[9] A. Fourney, G. Bansal, H. Mozannar, C. Tan, E. Horvitz, et al. Magentic-One: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468*, 2024.

[10] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *Proc. NeurIPS*, 2023.

[11] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Proc. NeurIPS*, 2024.

[12] Anthropic. Model Context Protocol specification. https://modelcontextprotocol.io/, 2024.

[13] OpenHands Contributors. OpenHands: An open platform for AI software developers as generalist agents. https://github.com/OpenHands/OpenHands (formerly `All-Hands-AI`); agent architecture: https://github.com/OpenHands/software-agent-sdk, 2024.

[14] P. Gauthier. Aider: AI pair programming in your terminal. https://github.com/Aider-AI/aider, 2024.

[15] Mini-SWE-Agent Contributors. Mini-SWE-Agent: A minimal software engineering agent. https://github.com/mini-swe-agent/mini-swe-agent, 2024.

[16] E. Schluntz and B. Zhang. Building effective agents. *Anthropic Engineering Blog*, December 2024. https://www.anthropic.com/research/building-effective-agents.

[17] J. Hadfield, B. Zhang, K. Lien, F. Scholz, J. Fox, and D. Ford. How we built our multi-agent research system. *Anthropic Engineering Blog*, June 2025. https://www.anthropic.com/engineering/multi-agent-research-system.

[18] K. Aizawa, B. Zhang, Z. Witten, and others. Writing effective tools for AI agents. *Anthropic Engineering Blog*, September 2025. https://www.anthropic.com/engineering/writing-tools-for-agents.

[19] P. Rajasekaran, E. Dixon, C. Ryan, J. Hadfield, R. Ayub, H. Moran, C. Rueb, and C. Jennings. Effective context engineering for AI agents. *Anthropic Engineering Blog*, September 2025. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents.

[20] Mistral AI. Mistral Vibe: A coding agent built on Mistral models. https://github.com/mistralai/mistral-vibe (formerly `mistralai/vibe`), 2026. Version 2.19.1, accessed July 2026.

[21] Google. Gemini CLI: An open-source AI agent that brings the power of Gemini to the terminal. https://github.com/google-gemini/gemini-cli, 2026. Version 0.50.0, accessed July 2026; consumer service retired June 2026 in the announced transition to the closed-source Antigravity CLI.

[22] Nous Research. Hermes Agent: The agent that grows with you. https://github.com/NousResearch/hermes-agent, 2026. Version v2026.7.7.2 (0.18.2), accessed July 2026.

[23] M. Zechner. Pi: The coding-agent harness you can make your own. https://github.com/earendil-works/pi, 2026. Version 0.80.6, accessed July 2026.

[24] Anomaly (formerly SST). OpenCode: The AI coding agent built for the terminal. https://github.com/anomalyco/opencode (formerly `sst/opencode`), 2026. Version 1.17.18, accessed July 2026.

[25] Databricks. Introducing Omnigent: A meta-harness to combine, control and share your agents. https://github.com/omnigent-ai/omnigent, June 2026. Version 0.4.0, accessed July 2026.

[26] xAI. Grok Build: xAI's coding agent CLI. https://x.ai/cli, May 2026. Accessed July 2026.

[27] Claw Code Contributors. Claw Code: A clean-room, provider-agnostic reimplementation of the Claude Code architecture. https://claw-code.codes/, 2026. Accessed July 2026.

[28] V. Trivedy. The anatomy of an agent harness. *LangChain Blog*, March 2026. https://www.langchain.com/blog/the-anatomy-of-an-agent-harness.

[29] The Deep Feed. Agent harness engineering: The new discipline, decoded. https://www.thedeepfeed.ai/posts/2026-05-09-agent-harness-engineering-the-discipline/, May 2026.

[30] B. Böckeler. Harness engineering for coding agent users. *martinfowler.com*, April 2026. https://martinfowler.com/articles/harness-engineering.html.

[31] P. Steinberger. "You shouldn't be prompting coding agents anymore. You should be designing loops that prompt your agents." Post on X, June 2026.

[32] A. Osmani. Loop engineering. https://addyosmani.com/blog/loop-engineering/, June 2026.

[33] S. O. de Macedo. What makes a harness a harness: Necessary and sufficient conditions for an agent harness. *arXiv preprint arXiv:2606.10106*, June 2026.

[34] B. Rombaut. Inside the scaffold: A source-code taxonomy of coding agent architectures. *arXiv preprint arXiv:2604.03515*, April 2026.

[35] HARBOR Authors. HARBOR: Automated harness optimization. *arXiv preprint arXiv:2604.20938*, April 2026.

[36] Code-as-Harness Authors. Code as agent harness. *arXiv preprint arXiv:2605.18747*, May 2026.

[37] LangChain. Deep Agents: Overview. https://docs.langchain.com/oss/python/deepagents/overview, 2026. Accessed July 2026.

[38] Anthropic. Claude Agent SDK: Build production AI agents with Claude Code as a library. https://code.claude.com/docs/en/agent-sdk/overview, 2026. Accessed July 2026.

[39] Pydantic Services Inc. Pydantic AI Harness. https://github.com/pydantic/pydantic-ai-harness, 2026. Version 0.6.0, accessed July 2026.

[40] Strands Agents. Harness SDK. https://github.com/strands-agents/harness-sdk, 2026. Accessed July 2026.

[41] Agent Client Protocol Working Group. Agent Client Protocol (ACP) specification. https://agentclientprotocol.com/, 2025.

[42] Google. Agent-to-Agent (A2A) Protocol specification. https://a2a-protocol.org/, 2025.

[43] Agent Skills Working Group. Agent Skills specification: file-system convention for capability bundles. https://agentskills.io/, 2025.

[44] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proc. ISSTA*, 2024. https://arxiv.org/abs/2404.05427.

[45] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, et al. OpenHands: An open platform for AI software developers as generalist agents. In *Proc. ICLR*, 2025. https://arxiv.org/abs/2407.16741.

[46] D. Chen, S. Lin, M. Zeng, D. Zan, et al. CodeR: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024. https://arxiv.org/abs/2406.01304.

[47] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. In *Proc. NeurIPS*, 2024. https://arxiv.org/abs/2403.17927.

[48] D. Arora, A. Sonwane, N. Wadhwa, A. Mehrotra, S. Utpala, R. Bairi, A. Kanade, and N. Natarajan. MASAI: Modular architecture for software-engineering AI agents. *arXiv preprint arXiv:2406.11638*, 2024. https://arxiv.org/abs/2406.11638.

[49] Y. Ma, R. Cao, Y. Cao, Y. Zhang, J. Chen, Y. Liu, Y. Liu, B. Li, F. Huang, and Y. Li. Lingma SWE-GPT: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024. https://arxiv.org/abs/2411.00622.

[50] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, et al. ChatDev: Communicative agents for software development. In *Proc. ACL*, 2024. https://arxiv.org/abs/2307.07924.

[51] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. https://arxiv.org/abs/2407.01489.

[52] A. Antoniades, A. Örwall, K. Zhang, Y. Xie, A. Goyal, and W. Wang. SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *Proc. ICLR*, 2025. https://arxiv.org/abs/2410.20285.

[53] K. Zhang, W. Yao, Z. Liu, Y. Feng, et al. Diversity empowers intelligence: Integrating expertise of software engineering agents. *arXiv preprint arXiv:2408.07060*, 2024. https://arxiv.org/abs/2408.07060.

[54] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *Proc. NeurIPS*, 2023. https://arxiv.org/abs/2303.11366.

[55] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, et al. Self-refine: Iterative refinement with self-feedback. In *Proc. NeurIPS*, 2023. https://arxiv.org/abs/2303.17651.

[56] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, N. Duan, and W. Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *Proc. ICLR*, 2024. https://arxiv.org/abs/2305.11738.

[57] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. In *Proc. ICLR*, 2024. https://arxiv.org/abs/2304.05128.

[58] Y. Qin, S. Liang, Y. Ye, K. Zhu, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *Proc. ICLR*, 2024. https://arxiv.org/abs/2307.16789.

[59] Y. Du, F. Wei, and H. Zhang. AnyTool: Self-reflective, hierarchical agents for large-scale API calls. In *Proc. ICML*, 2024. https://arxiv.org/abs/2402.04253.

[60] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proc. EMNLP*, 2023. https://arxiv.org/abs/2303.12570.

[61] T. Liu, C. Xu, and J. McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *Proc. ICLR*, 2024. https://arxiv.org/abs/2306.03091.

[62] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, et al. CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion. In *Proc. NeurIPS Datasets and Benchmarks*, 2023. https://arxiv.org/abs/2310.11248.

[63] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried. CodeRAG-Bench: Can retrieval augment code generation? In *Findings of NAACL*, 2025. https://arxiv.org/abs/2406.14497.

[64] E. Bogomolov, A. Eliseeva, T. Galimzyanov, E. Glukhov, et al. Long Code Arena: A set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612*, 2024. https://arxiv.org/abs/2406.11612.

[65] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, et al. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In *Proc. EMNLP System Demonstrations*, 2024. https://arxiv.org/abs/2402.16667.

[66] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023. https://arxiv.org/abs/2309.07864.

[67] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang. Large language model based multi-agents: A survey of progress and challenges. In *Proc. IJCAI Survey Track*, 2024. https://arxiv.org/abs/2402.01680.

[68] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou. Large language model-based agents for software engineering: A survey. *ACM Trans. Softw. Eng. Methodol.*, 2025. To appear. https://arxiv.org/abs/2409.02977.

[69] X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, et al. AgentBench: Evaluating LLMs as agents. In *Proc. ICLR*, 2024. https://arxiv.org/abs/2308.03688.

[70] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Proc. NeurIPS*, 2023. https://arxiv.org/abs/2305.10601.

[71] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proc. ACL*, 2023. https://arxiv.org/abs/2305.04091.

[72] Y. Zhang, R. Sun, Y. Chen, T. Pfister, R. Zhang, and S. Ö. Arik. Chain of agents: Large language models collaborating on long-context tasks. In *Proc. NeurIPS*, 2024. https://arxiv.org/abs/2406.02818.

[73] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proc. ICML*, 2024. https://arxiv.org/abs/2310.04406.

[74] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. https://arxiv.org/abs/2310.08560.

[75] H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proc. EMNLP*, 2023. https://arxiv.org/abs/2310.05736.

[76] E. Debenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Proc. NeurIPS Datasets and Benchmarks*, 2024. https://arxiv.org/abs/2406.13352.

[77] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proc. AISec*, 2023. https://arxiv.org/abs/2302.12173.

[78] T. Yuan, Z. He, L. Dong, Y. Wang, et al. R-Judge: Benchmarking safety risk awareness for LLM agents. In *Findings of EMNLP*, 2024. https://arxiv.org/abs/2401.10019.

[79] J. Ye, S. Li, G. Li, C. Huang, S. Gao, Y. Wu, Q. Zhang, T. Gui, and X. Huang. ToolSword: Unveiling safety issues of large language models in tool learning across three stages. In *Proc. ACL*, 2024. https://arxiv.org/abs/2402.10753.

[80] A. Mitra, L. Del Corro, G. Zheng, S. Mahajan, D. Rouhana, et al. AgentInstruct: Toward generative teaching with agentic flows. *arXiv preprint arXiv:2407.03502*, 2024. https://arxiv.org/abs/2407.03502.

[81] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, and L. Zhong. Prompt cache: Modular attention reuse for low-latency inference. In *Proc. MLSys*, 2024. https://arxiv.org/abs/2311.04934.

[82] L. Chen, M. Zaharia, and J. Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Trans. Mach. Learn. Res.*, 2024. https://arxiv.org/abs/2305.05176.

[83] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica. RouteLLM: Learning to route LLMs with preference data. In *Proc. ICLR*, 2025. https://arxiv.org/abs/2406.18665.

[84] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee, V. Rühle, L. V. S. Lakshmanan, and A. H. Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *Proc. ICLR*, 2024. https://arxiv.org/abs/2404.14618.

[85] X. Lin, C. Ruan, B. Rozière, M. Tufano, M. Velez, and B. Shen. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, April 2026.

[86] H. Guo, Y. Hao, Y. Zhang, et al. A measurement study of Model Context Protocol ecosystem. *arXiv preprint arXiv:2509.25292*, 2025. https://arxiv.org/abs/2509.25292.

[87] R. Xu and Y. Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026. https://arxiv.org/abs/2602.12430.

[88] S. Saha and P. Hemanth. Skilldex: A package manager and registry for agent skill packages with hierarchical scope-based distribution. *arXiv preprint arXiv:2604.16911*, 2026. https://arxiv.org/abs/2604.16911.

[89] Z. Guo, Z. Chen, et al. SkillProbe: Security auditing for emerging agent skill marketplaces via multi-agent collaboration. *arXiv preprint arXiv:2603.21019*, 2026. https://arxiv.org/abs/2603.21019.

[90] S. Kapoor, N. Kolt, and S. Lazar. Resist platform-controlled AI agents and champion user-centric agent advocates. In *Proc. ICML*, 2025. https://arxiv.org/abs/2505.04345.

[91] R. Robbes, T. Matricon, et al. Agentic much? Adoption of coding agents on GitHub. *arXiv preprint arXiv:2601.18341*, 2026. https://arxiv.org/abs/2601.18341.

[92] H. Li, H. Zhang, and A. E. Hassan. The rise of AI teammates in Software Engineering 3.0: How autonomous coding agents are reshaping software engineering. *arXiv preprint arXiv:2507.15003*, 2025. https://arxiv.org/abs/2507.15003.