Agent and Harness Architecture

Last modified: 2026-09-13 00:11:34 (PDT)

Behind every coding agent is a system architecture: a model, an execution loop, tool definitions, and a *harness* that manages context, permissions, and session state. This chapter covers how coding agents and harnesses are structured, how they run under the hood, and how the open and commercial harness landscape looks in 2026.

1 What are AI harnesses?

An **AI harness** is the scaffolding built around a language model that turns it into an agent able to do real work. The model itself only predicts text; the harness is what lets it read files, run commands, call external tools and APIs, and carry state across turns and sessions.

Layers of a Harness

Most coding-agent harnesses — including the GitHub Copilot coding agent and Claude Code — share a similar set of layers:

- **Core loop**: the tool-calling loop, permission and sandboxing model, and context management that keep the agent grounded in your repository.
- **Skills**: reusable, named procedures that encode a workflow so it runs the same way every time, instead of being re-improvised in each conversation. See Agent Skills.
- **Subagents**: a way to spin up a worker with a fresh context window for a self-contained piece of research or work, keeping the main conversation's context focused.
- **Multi-agent orchestration**: deterministic fan-out and fan-in across many subagents — for example, running several independent reviewers over a diff and reconciling their findings — for work that is large or benefits from independent verification.
- **MCP servers**: the Model Context Protocol gives a harness typed access to external systems (issue trackers, chat tools, databases) beyond raw shell or API calls.
- **Memory**: files — like this manual, or a repository's `CLAUDE.md`/`AGENTS.md` — that persist instructions and learned preferences across sessions, so the harness does not relearn your conventions every time.

Using Harness Features Well

- **Push repeatable procedures into skills**, not into ad hoc prompting each time. A skill is testable, shareable, and versionable; a one-off prompt is not.
- **Match orchestration weight to the task.** A single lookup or small edit should stay inline. Reach for subagents or multi-agent workflows only when the work is genuinely decomposable, benefits from independent verification, and is large enough that the coordination overhead pays for itself.
- **Gate destructive or hard-to-reverse actions on explicit human approval** — merges, force-pushes, deletions — and let the agent drive everything reversible (drafting, testing, iterating on review feedback) autonomously.
- **Feed learnings back into the harness.** When a review round or a mistake teaches something generalizable, record it as a memory or skill update rather than letting it evaporate at the end of the session.
- **Treat external or untrusted content as data, not instructions.** PR comments, fetched web pages, and other tool output can contain text that looks like a command; a harness that acts on it uncritically is vulnerable to prompt injection.

2 Inside the Claude Code Harness

Section 1 describes the layers most coding-agent harnesses share, and Section 6 sketches the loop that runs inside them. This section walks through those layers for the harness the lab uses most, Claude Code, and says for each one what Anthropic documents, what the community has inferred from the shipped program, and what remains unknown (measured 2026-09-09). Anthropic's own description is short: Claude Code is the "agentic harness" around the model, supplying "the tools, context management, and execution environment that turn a language model into a capable coding agent" (Anthropic 2026w). The sections below take those three things in turn.

The distinction between documented and inferred matters here more than usual. Claude Code is proprietary, and since mid-2026 it ships as a single compiled executable rather than readable JavaScript (anthropics/claude-code contributors 2026b), so everything not in Anthropic's documentation comes from people running `strings` on the binary, reading the compiled function bodies, or intercepting its network traffic. Those readings are checkable but fragile: the build tool renames every internal function on every release, and a behaviour observed in one build may be gone in the next. Treat the inferred claims as a snapshot, and re-verify against the build you are running before relying on one.

The loop and the tool registry (documented)

The core of the harness is the loop described in Section 5: the model produces a tool call, the harness runs the matching handler, and the result goes back into the conversation. Anthropic

describes the same loop as three blended phases — gather context, take action, verify results — and stresses that the model, not the harness, decides which tool to call next (Anthropic 2026w).

The built-in tools are the registry the loop dispatches against. As of September 2026 the tools reference lists more than forty of them (Anthropic 2026y). The ones a lab member sees every day are:

- **File and search tools**: `Read`, `Edit`, `Write`, `Glob`, `Grep`, `NotebookEdit`, and `LSP` for language-server code intelligence.
- **Execution tools**: `Bash`, `PowerShell`, and `Monitor`, which streams a background command's output lines back to the model.
- **Orchestration tools**: `Agent` (spawns a subagent), `Skill` (runs a skill), `SendMessage` and `ListAgents` (agent teams), the `Task*` family (a session task list), `CronCreate` and `ScheduleWakeup` (timers), and `ToolSearch` (loads deferred tool schemas on demand).
- **Interaction tools**: `AskUserQuestion`, `EnterPlanMode` and `ExitPlanMode`, `EnterWorktree` and `ExitWorktree`, `Artifact`, and `SendUserFile`.

Each tool is a schema the model sees plus a handler it does not, exactly as Section 4 describes. The tool names double as the vocabulary of the permission system, hook matchers, and subagent tool lists, so a rule written as `Bash(git *)` refers to the same `Bash` entry the model calls (Anthropic 2026y). MCP servers (below) extend this registry without changing its shape.

The system prompt: what is public and what is not

Anthropic does not publish Claude Code's system prompt. What the documentation does describe is the *startup context*: everything already in the model's window before you type a word. The context-window page walks through it in order (Anthropic 2026u):

- the system prompt itself, including an output style and any `--append-system-prompt` text, which "both go into the system prompt the same way";
- an environment block (working directory, platform, shell, whether this is a git repository), with git branch, status, and recent commits loaded "as a separate block at the very end of the system prompt";
- MCP tool *names*, with their full schemas deferred until `ToolSearch` loads them;
- one-line skill descriptions, so the model knows what it can invoke;
- your `CLAUDE.md` files and the first 200 lines of auto memory.

One documented detail changes how the rest of this section reads: `CLAUDE.md` content "is delivered as a user message after the system prompt, not as part of the system prompt itself" (Anthropic 2026x). So the layered picture is harness-authored system prompt first, then your instructions, then the conversation. The same page is candid that the harness "treats them as context, not enforced configuration": a `CLAUDE.md` instruction is prose the model may weigh

against other prose, and anything that must hold regardless goes in a permission rule or a hook.

Three documented levers reach the system prompt directly (Anthropic 2026s):

- `--append-system-prompt` adds text to it for one invocation.
- `CLAUDE_CODE_SIMPLE_SYSTEM_PROMPT=1` requests "a shorter system prompt and abbreviated tool descriptions", and `0` opts out "even on models where the experiment or server configuration would otherwise enable it" — the documentation's own acknowledgment that the prompt varies by model and by server-side configuration.
- `--bare` (equivalently `CLAUDE_CODE_SIMPLE=1`) runs "with a minimal system prompt and only the Bash, file read, and file edit tools", and disables discovery of hooks, skills, subagents, plugins, MCP servers, auto memory, and `CLAUDE.md`. It is a stripped harness, not a stripped prompt, which is why it is the wrong fix for the problem described next.

What the community has inferred is the prompt's internal structure. Binary analysis reported in the Claude Code issue tracker shows the prompt assembled from named sections, each behind a function that returns text or `null`, with names such as `anti_verbosity`, `thinking_guidance`, `action_caution`, `delivering_work_max`, `overcorrection`, `subagent_steer_delegation`, `heron_brook`, and `autonomy_append` (anthropics/claude-code contributors 2026a, 2026b). Some sections are gated on a *model capability* declared in an internal model registry (the reported gate for Opus 5 is a capability string `opus_5_prompt_bundle`), some on a remote feature flag, and some on both. One analyst counted roughly 23 sections in a mid-2026 build, with user memory rendered around eighth and `heron_brook` second-to-last (anthropics/claude-code contributors 2026b). None of this is documented, the counts and names are build-specific, and the section order is one person's inference from the compiled code — so the safe statement is that the system prompt is *modular and conditional*, not that any particular section exists on your machine today.

Hard-coded instructions, remote flags, and `heron_brook`

`heron_brook` is not a feature. It is the internal name of one system-prompt section, and its story is the clearest public window into how the harness's prompt is controlled. The name follows the pattern of every Claude Code feature flag: a `tengu_` prefix (the product's internal code name, per one community catalog of the binary's strings (wtfwhs 2026)) followed by an auto-generated `adjective_noun` pair that carries no meaning of its own. `tengu_fennel_godwit`, mentioned below, is the same kind of name. Anthropic's only public statement touching the section is the short reply quoted below.

May 2026: the slot appears. Claude Code 2.1.150 shipped with a changelog entry reading, in full, "Internal infrastructure improvements (no user-facing changes)" (Anthropic 2026n). Within days a user reported that the build added a function reading a string from two network-backed sources — the `client_data` field of the `/api/claude_cli/bootstrap` response, and a GrowthBook feature flag named `tengu_heron_brook` that refreshes every 60 seconds — and

inserting it verbatim as a system-prompt section (anthropics/claude-code contributors 2026a). The same finding reached Hacker News under the title "Claude Code now allows Anthropic to remotely inject system prompts" (matheusmoreira 2026). An Anthropic engineer replied on the issue that the company "sometimes run[s] experiments on changes to our system prompt so that we can evaluate how a change impacts quality before fully rolling it out", that users can opt out with `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC=1` and `DISABLE_GROWTHBOOK=1`, and that Claude Code should not be used through an untrusted proxy (anthropics/claude-code contributors 2026a). The issue was closed. A separate HackerOne report on the same channel was closed as *Informative*, with Anthropic's security team stating that TLS is the integrity boundary and no response signing is planned (cnighswonger 2026). An independent catalog of the binary's strings describes the section the same way: "a server-controlled prompt-injection slot" (wtfwhs 2026).

July 2026: the slot gets a hard-coded default. Claude Code 2.1.219 was the release that added Opus 5 (Anthropic 2026n). Users on that model noticed their sessions had stopped delegating to subagents, and the model, when asked, quoted two lines it said it had been given:

> Do not call the `AgentTool` unless the user requested it Do not use workflows or deep-research unless the user requested it

The canonical report (anthropics/claude-code contributors 2026b) traced the lines to a constant compiled into the binary, served as the *fallback* text of the `heron_brook` section when neither the bootstrap response nor the GrowthBook flag supplies a string. The reported gate has two parts: the section fires only for models whose registry entry carries the `opus_5_prompt_bundle` capability (in the builds examined, only `claude-opus-5`, not the Fable, Sonnet, or Haiku families), and only while a kill-switch flag, `tengu_fennel_godwit`, is false, which is its default. A thread in the r/ClaudeCode community on Reddit carried the finding to a wider audience (r/ClaudeCode 2026). (Reddit blocks unauthenticated fetches, so the thread is cited by title only.)

Four consequences are reported across that issue and its siblings:

- **It overrides user configuration silently.** Users whose `CLAUDE.md` *requires* delegation — including several with mandatory review-by-subagent gates — saw zero subagent dispatches for whole sessions, and one fleet operator measured Opus 5 sessions taking 2.5 times as many assistant turns as Opus 4.8 on the same work, which they attributed to serial work that had previously fanned out (anthropics/claude-code contributors 2026b). The documentation says delegation is driven by each subagent's `description` (Anthropic 2026q); a separate report argues the directive contradicts that documented behaviour (anthropics/claude-code contributors 2026e).
- **The model attributes the line to you.** Because the section arrives in the same voice as everything else, and because `CLAUDE.md` is a *user* message that appears earlier, the model reads "unless the user requested it" as the user's own standing rule and tells

users their configuration forbids delegation when it says the opposite (anthropics/claude-code contributors 2026d). A related report frames the underlying gap: there is no defined precedence between an Anthropic-authored prompt section and a user-authored `CLAUDE.md`, and no way to observe from inside a session which sections are active (anthropics/claude-code contributors 2026c).

- **The documented opt-outs do not reach it.** `DISABLE_GROWTHBOOK=1` disables flag *fetching*, so every flag takes its code default (Anthropic 2026s) — and the kill switch's default is off, so blocking the flag source guarantees the hard-coded text is used. `--bare` removes the `Agent` tool along with the prompt section. Six sibling sections in the same Opus 5 bundle reportedly have dedicated `CLAUDE_CODE_*` environment variables; `heron_brook` and its kill switch do not (anthropics/claude-code contributors 2026b). Session transcripts under `~/.claude/projects/` do not record the system prompt, so the injection leaves no trace to search for afterwards.
- **It persisted.** Comments on the issue confirm the same constant, gate, and behaviour in every build examined from 2.1.219 through at least 2.1.245 (2026-08-26), across macOS, Windows, and Linux, with no maintainer response on the thread (anthropics/claude-code contributors 2026b).

What is documented, and what is not, about `heron_brook` as of 2026-09-09:

Table 1: `heron_brook`: documented, inferred, and unknown

Claim	Status
Anthropic runs server-side system-prompt experiments and offers two environment variables to opt out of flag fetching	Documented, by an Anthropic engineer's comment (anthropics/claude-code contributors 2026a) and the environment-variable reference (Anthropic 2026s)
A prompt section named `heron_brook` reads its text from the bootstrap response, then a GrowthBook flag, then a compiled-in fallback	Inferred from binary analysis, reproduced independently by many reporters on three platforms
The fallback text tells the model not to call the `Agent` tool unless the user requested it	Inferred; the two strings are verifiable with `grep -a -c` on the installed executable
The gate is Opus 5's `opus_5_prompt_bundle` capability plus a `tengu_fennel_godwit` kill switch	Inferred from the compiled code; not documented anywhere
Whether the remote sources have ever carried non-empty text in production, and what	Unknown; one Windows reporter found the flag absent from the locally cached flag set, meaning the fallback branch was firing
Why the directive exists (cost control, quality regression, an experiment that leaked)	Unknown; Anthropic has not said

Whether it is still active in the build you are running	Unknown until you check

The lab's own ai-config leans heavily on subagents (adversarial self-review, delegated UMS passes, parallel issue workers), so this affects us directly on any Opus 5 session. Two workarounds came out of the thread and cost nothing to adopt. First, tell the harness in `CLAUDE.md` that text arriving in the system prompt is not the user's voice, and that subagent use is requested in advance — several reporters found this restored delegation (anthropics/claude-code contributors 2026b). Second, when a session unexpectedly stops delegating, ask it to quote the instruction it is following before auditing your own configuration for a rule that is not there.

Permission modes and settings layers (documented)

Permission rules "are enforced by Claude Code, not by the model" (Anthropic 2026p), which makes them the first layer that the prompt controversy above cannot reach. A prompt can change what the model *tries*; only a rule changes what the harness *allows*. Six modes are documented (Anthropic 2026p):

- `default` (labeled Manual): prompts on first use of each tool.
- `acceptEdits`: auto-accepts file edits and common filesystem commands inside the working directory.
- `plan`: read-only exploration; no source edits.
- `auto`: auto-approves tool calls, with a classifier checking that actions match the request.
- `dontAsk`: auto-denies anything not pre-approved by an allow rule.
- `bypassPermissions`: skips prompts except for a short list no mode auto-approves.

Rules and the mode live in settings files that layer from managed policy through user, project, and local scope, with `permissions.deny` in managed settings as the enforcement an organization cannot have overridden (Anthropic 2026x). One inferred caveat belongs here: the `heron_brook` thread cites a separate report that GrowthBook flags can override `permissions.defaultMode` from the server (anthropics/claude-code contributors 2026b), so even this layer may have a remotely controlled input. That report is cited second-hand and was not verified for this section.

Hooks (documented)

Hooks are the harness's event system: shell commands, HTTP endpoints, MCP tool calls, single-turn prompts, or spawned subagents that run at fixed lifecycle points (Anthropic 2026v). The reference lists more than thirty events, grouped as session (`SessionStart`, `SessionEnd`), per-turn (`UserPromptSubmit`, `Stop`), tool execution (`PreToolUse`, `PostToolUse`, `PermissionRequest`), agent (`SubagentStart`, `SubagentStop`), context (`InstructionsLoaded`, `PreCompact`, `PostCompact`), and several more. A `PreToolUse` hook that exits with code 2

blocks the call, and a hook's JSON output can deny, add context, or rewrite the tool's input. This is the mechanism customizing an agent calls "the part the model cannot talk its way around", and it is the right home for any rule that must hold regardless of what the prompt says — including a rule that the prompt itself has been told to contradict. `InstructionsLoaded` is also the only documented way to log exactly which instruction files a session loaded and when (Anthropic 2026x).

Subagents (documented)

The `Agent` tool spawns a subagent: a fresh instance of the same loop with its own context window, its own system prompt, a restricted tool list, and independent permissions (Anthropic 2026q). Built-in subagents include `Explore` and `Plan` (read-only; they skip `CLAUDE.md` and git status to stay cheap) and `general-purpose`; custom ones are markdown files with front matter in `.claude/agents/` or `~/.claude/agents/`, as Section 4 illustrates, and plugins can ship more. The harness watches those directories and picks up edits without a restart. Delegation is documented as description-driven: "Claude uses each subagent's description to decide when to delegate tasks", and the combined descriptions are capped at 15,000 tokens before a startup warning (Anthropic 2026q). Delegation can be removed entirely by denying the `Agent` tool in `permissions.deny`. That is the documented, user-controlled way to stop subagent use; `heron_brook` is the undocumented, server-controlled one.

Skills and plugins (documented)

A skill is a `SKILL.md` with front matter and a body, run through the built-in `Skill` tool rather than as a tool of its own (Anthropic 2026y). Only skill *descriptions* sit in the startup context; the body loads when the skill is invoked, and a skill marked `disable-model-invocation: true` stays out of context entirely until you type its slash command (Anthropic 2026u). Agent Skills covers the format. A plugin bundles skills, agents, hooks, MCP and LSP server configurations, background monitors, and default settings into one directory with a `.claude-plugin/plugin.json` manifest, namespaces its skills as `/plugin-name:skill`, and can be loaded from a marketplace, from a local path with `--plugin-dir`, or from the user's skills directory (Anthropic 2026r). Plugins deep dive dissects the bundle, and how config reaches a machine describes how the lab's own plugin reaches a machine.

MCP servers (documented)

An MCP server adds tools to the registry without changing the loop. By default the harness lists only the server's tool *names* at startup and defers the full schemas, loading each on demand through `ToolSearch`; `ENABLE_TOOL_SEARCH=false` loads everything up front (Anthropic 2026u). That deferral is why a session can carry dozens of servers without paying their schema cost on every turn. Two dedicated tools, `ListMcpResourcesTool` and `ReadMcpResourceTool`,

expose server resources as well as tools (Anthropic 2026y). MCP server setup covers registration and its failure modes.

Memory and `CLAUDE.md` loading (documented)

Two mechanisms carry knowledge across sessions (Anthropic 2026x):

- **`CLAUDE.md` files**, loaded in a fixed order: managed policy, then `~/.claude/CLAUDE.md`, then every `CLAUDE.md` and `CLAUDE.local.md` from the filesystem root down to the working directory, concatenated rather than overriding each other. Files in subdirectories load on demand when the model reads files there. `@path` imports expand at launch to a depth of four, `.claude/rules/*.md` files load alongside, with `paths:` front matter scoping a rule to matching files, and block-level HTML comments are stripped before injection. A file over `4 MiB` is skipped.
- **Auto memory**, notes the model writes itself under `~/.claude/projects/<project>/memory/`, of which the first 200 lines or 25 KB of `MEMORY.md` load every session and topic files load on demand.

`/context` shows which files actually loaded, and `/memory` opens them. The "delivered as a user message" detail above applies to all of it.

Compaction (documented)

When the context window nears its limit, the harness summarizes the conversation and replaces it; `/compact` does the same on demand, optionally with a focus instruction, and `/autocompact <tokens>` moves the threshold (Anthropic 2026u). What survives is specific:

- the system prompt and output style are unchanged, because they were never in message history;
- project-root `CLAUDE.md`, rules without a `paths:` field, auto memory, and a plan-mode plan are re-injected from disk;
- path-scoped rules and nested `CLAUDE.md` files reload only as matching files are read again;
- up to five recently modified files are re-read;
- invoked skill bodies are re-injected, capped at 5,000 tokens each and 25,000 total;
- context that hooks added earlier is summarized away, unless a `SessionStart` hook matching the `compact` source re-adds it;
- the skill *listing* is not reloaded (Anthropic 2026u).

The lab's `compress-session` practice exists because the automatic summary guesses what matters; running `/compact` with a focus before it triggers keeps that choice with you.

The Agent SDK relationship (documented)

The Claude Agent SDK "gives you the same tools, agent loop, and context management that power Claude Code, programmable in Python and TypeScript" (Anthropic 2026a). It is the harness as a library: built-in tools, hooks, subagents, MCP, permissions, sessions, skills, memory, and plugins are all listed as SDK capabilities, and it loads `.claude/` and `~/.claude/` configuration "same as Claude Code". Anthropic distinguishes it from the Client SDK (raw API access, where you write the loop yourself), from the CLI (interactive use), and from Managed Agents (a hosted product where Anthropic runs the sandbox). Other languages drive the same loop by running the CLI as a subprocess with `-p` and `--output-format json` (Anthropic 2026a). The corollary for this section is that everything above, including the modular and conditional system prompt, is what an SDK-built agent inherits. The SDK documents a way to replace or extend the system prompt, which is the supported route where the CLI offers only `--append-system-prompt`.

Summary: what you can and cannot see

Claude Code's execution engine, tool registry, permission system, hooks, subagents, skills, plugins, MCP integration, memory loading, compaction, and SDK surface are all documented in detail, and the documentation is unusually precise about what loads when. The one layer it does not describe is the system prompt, and the `heron_brook` episode shows that layer is modular, model-gated, and partly remote-controlled, with no in-session way to observe which parts are active. The practical rule for the lab follows from the documentation's own advice: anything that must hold goes in a permission rule or a hook, which the harness enforces, not in `CLAUDE.md`, which the model weighs against text you cannot see.

3 How Agents Are Structured and Implemented

An **agent** is not part of the harness itself. It is a configuration — a goal, a role, a bounded toolset, and a stopping condition — executed on top of the harness's core loop (see Section 1). A single harness can host many different agents at once: a main conversation, and any number of subagents it spawns.

The Shape of an Agent

Structurally, an agent is a small record plus a fresh execution of the harness's loop:

- **Identity**: a name and description, used to route a task to the right agent ("when should this agent be picked?").
- **Instructions**: a system-prompt fragment that specializes behavior, for example "you are a read-only search agent."

- **Tool allowlist**: a subset of the harness's tool registry this agent may call — often narrower than the caller's own toolset.
- **Model and effort**: which model backs the agent, and how much reasoning depth it applies; these can differ from the caller's own settings.
- **Output contract**: whether the agent returns free text, or must call a schema-validated tool to return a typed result.

How an Agent Runs

1. **Spawn**: allocate a fresh message history with no inherited conversation — just the agent's instructions, plus whatever prompt the caller wrote. A subagent prompt needs to be self-contained for this reason: brief it like a colleague who just walked into the room.
2. **Run**: execute the harness's core loop (model call, parse tool calls, execute against the allowlist, append results, repeat), the same machinery the main session uses, just bound to a narrower toolset and a different system prompt.
3. **Terminate**: stop when the model emits a final answer with no further tool calls, when a schema-validated call satisfies the output contract, when it hits an error or a budget ceiling, or when the caller kills it.
4. **Return**: everything that happened inside the agent — every tool call, every intermediate step — is discarded from the caller's context. Only the final text or validated object crosses back. This is the point of an agent: it is a context-isolation boundary, not just a prompt.

Composability and Its Limits

Agents can spawn agents: an orchestration layer runs many agent instances, some concurrently, and composes their results. Nesting is deliberately capped, usually to one level, because unbounded recursion has no natural stopping point and burns cost and time with no guardrail. An orchestration script is a scheduler over independent agent-loop instances, not a different execution model.

Two Axes That Define an Agent's Behavior

- **Isolation versus continuation**: a subagent gets no inherited context (isolation); a resumed agent keeps its own accumulated history and continues it (continuation). Both use the same loop machinery, differing only in history-management policy.
- **Free-form versus structured output**: by default an agent returns prose. Given a schema, it is forced to call a structured-output tool instead, turning it into a typed function from the caller's point of view — input in, validated object out — even though internally it is still a multi-turn loop.

4 How Harnesses and Agents Are Built

The layers described above are not all built the same way. Some are ordinary software; others are just text files the harness reads at runtime.

The Execution Engine Is Ordinary Software

The program that runs the core loop — calling the model, parsing tool calls, enforcing permissions, managing the sandbox — is compiled or interpreted source code, the same as any other application. There is no markdown involved here; this layer is what makes a harness a harness, rather than just a prompt someone wrote.

Agent and Skill Definitions Are Markdown with a Front Matter Header

An agent's identity, and a skill's metadata, are usually just a markdown file with a YAML front matter header. For example, a custom Claude Code subagent defined in `.claude/agents/code-reviewer.md`:

```
---
name: code-reviewer
description: Reviews diffs for bugs and style issues.
tools: Read, Grep, Glob
model: sonnet
---

You are a meticulous code reviewer. Focus on correctness,
security, and idiomatic style.
```

The front matter is parsed as structured configuration (name, description, allowed tools, model); the markdown body below it becomes that agent's system prompt, verbatim. An Agent Skill's `SKILL.md` follows the same shape: front matter for discovery metadata, a markdown body for instructions, and an optional folder of bundled scripts or reference files alongside it. No compilation step is involved; the harness reads the file and uses it directly.

Tools Are a Schema Paired with a Handler

A tool definition has two parts: a JSON Schema describing its parameters, which is the only part the model ever sees, and a handler function, ordinary code that performs the actual action (reading a file, running a command, calling an API). The schema is declarative data; the handler is real software the model never inspects or writes.

Orchestration Needs Real Code

Multi-agent orchestration cannot be expressed declaratively, because it needs genuine control flow — loops, conditionals, parallel fan-out with a concurrency limit. So orchestration scripts are literal source files, executed by the harness, not parsed as prompt text the way an agent definition is.

Memory Is Just Prose

Files like this manual, or a repository's `CLAUDE.md`/`AGENTS.md`, carry no front matter and no schema. They are concatenated into the system prompt as plain text, and the harness trusts the model to read and follow that prose, the same way it follows any other instruction in its context.

5 What Kind of Program Is an Agent?

An agent is not a standalone program that does the reasoning itself. It is an **orchestration** program: something closer in shape to a chat client or a build tool than to a compiler or a web server.

It Is I/O-Bound, Not Compute-Bound

The actual token prediction happens on remote inference infrastructure, reached over HTTPS. The agent process itself does no heavy computation; it spends almost all its wall-clock time waiting — for a model API response, for a shell command to finish, for a file read. Structurally it is an **event-loop** program, the same category as a network client. Its core loop can be sketched in a few lines:

```
# Start the conversation with the agent's instructions and the task.
history = [system_prompt, user_message]

while True:
    # Send everything so far to the model, along with what it's allowed to call.
    response = call_model(history, tools=tool_schemas)
    history.append(response)

    # No tool calls means the model gave a final answer -- stop.
    if not response.tool_calls:
        break

    # Otherwise, run each requested tool and feed the result back in,
```

```
    # so the next model call can see what happened.
    for call in response.tool_calls:
        result = tool_registry[call.name](call.arguments)
        history.append(result)
```

Everything a harness adds — permissions, sandboxing, memory, subagents — is scaffolding wrapped around this loop, not a replacement for it.

Real Open-Source Examples

Because most production coding-agent harnesses are closed source, the clearest way to see this shape in real code is to read an open-source one:

- **aider** — an open-source AI pair-programming CLI.
- **SWE-agent** — a research coding-agent harness from Princeton NLP, described in its associated paper.
- **OpenHands** (formerly OpenDevin) — a general-purpose open-source agent platform.

Their orchestration code runs to thousands of lines, because that is where the real engineering lives: retries, streaming, permission checks, and state management. A single *agent definition* running on top of that engine, by contrast, is typically tens of lines (see Section 4).

Where It Runs

- **The harness process**: an ordinary OS process, either on your own machine (CLI mode) or inside an ephemeral, managed cloud container (remote/web mode) that is discarded when the session ends.
- **Subagents**: run inside the *same* host process as their caller, not a separate container. They differ only in having their own message history and a narrower tool set, unless a workflow explicitly asks for a separate git worktree to avoid file conflicts during parallel edits.
- **The model call itself**: not part of the agent's environment at all. It is a network request to inference infrastructure the agent has no visibility into, beyond the request and response.

So an agent's lifetime is scoped to a single task, not persistent: it starts when given a goal, runs for as long as its loop keeps producing tool calls, and ends the moment a stopping condition fires.

6 How Does a Harness Relate to an Agent?

The relationship between a harness and an agent is closer to an **interpreter** running a program than to two peers calling each other.

Does the Harness Call the Agent, or the Agent Call the Harness?

Harness to agent: not a call, an instantiation. The harness does not "call" an agent as a subroutine it invokes and waits on. An agent has no code of its own outside the harness's loop (see Section 5) — its whole behavior *is* that loop, running with the agent's configuration (instructions, tool allowlist, model) loaded in. The harness instantiates and runs an agent, start to termination; it is not a function call with a return address.

Agent to harness: yes, a real call, via tool calls. While an agent's loop is running, the model produces a tool-call request, and the harness's dispatcher looks up and executes the matching handler — read a file, run a command, call an API. So the concrete direction of calling is **agent calls harness**, through tool dispatch, not the reverse.

Agent to agent: routed through the harness. When a parent agent spawns a subagent, it does not call that subagent directly. It issues a tool call that the harness's dispatcher handles by spinning up a fresh instance of its own loop (see Section 3), running it to completion with the subagent's configuration, and handing the result back to the parent as a tool result. Even "agent calls agent" bottoms out as: parent calls harness, harness instantiates and runs a new agent, harness returns that agent's output to the parent.

Sketching the Harness's Own Loop

The agent loop sketched earlier is really just the innermost piece. The harness wraps a bootstrap step and a permission/dispatch layer around it:

```
def run_harness():
    # Load everything the loop will need before any conversation starts.
    tools = load_tool_registry()          # built-ins, plus whatever MCP servers expose
    memory = load_memory(CLAUDE_MD_PATHS) # CLAUDE.md / AGENTS.md, concatenated

    # The "main session" is just the harness's own loop, run with a default,
    # unrestricted configuration -- not a separate program.
    main_agent = Agent(config=default_config, system_prompt=memory)
    return run_agent(main_agent, tools)

def run_agent(agent, tools):
    history = [agent.system_prompt, agent.first_message]
    while True:
```

```
        response = call_model(history, tools=agent.tool_schemas)
        history.append(response)
        if not response.tool_calls:
            break
        for call in response.tool_calls:
            # Every tool call passes through the harness's own gate first,
            # regardless of which agent requested it.
            check_permission(call)

            if call.name == "spawn_subagent":
                # A subagent is not called directly -- the harness recurses
                # into a fresh instance of this same loop, then hands the
                # finished result back as an ordinary tool result.
                result = run_agent(Agent(call.arguments), tools)
            else:
                result = tools[call.name](call.arguments)

            history.append(result)
    return history[-1]
```

`run_agent` is identical in shape to the loop in Section 5. `run_harness` and the permission check are the parts that only exist at the harness level, not inside any individual agent. That recursive call — `run_agent` calling itself for a subagent — is the concrete mechanism behind "agent calls agent, routed through the harness," described in the previous subsection.

What Do You Launch When You Type `claude`?

Typing `claude` at a shell starts the harness process: it initializes the engine — the permission system, the tool registry, MCP client connections, and memory loaded from `CLAUDE.md`/`AGENTS.md` files. But the harness does not sit idle waiting for a program to be supplied separately. It immediately instantiates the **default agent** — the "main session" — to handle the interactive conversation: full tool access, a system prompt assembled from the loaded config, no restricted allowlist. That default agent is simply the harness's baseline configuration for its own loop, not a second thing launched afterward.

There is no observable moment of "harness running, no agent yet." The closest analogy is typing `python` at a shell: it launches the interpreter *and* drops you straight into a REPL evaluating your input, rather than leaving the interpreter idle with nothing loaded. The difference is that the harness's default "program" is built in (the main-session agent's configuration), rather than something you must supply. A custom subagent or a `.claude/agents/*.md` definition, by contrast, *is* a separate agent, instantiated on demand, mid-session, when the already-running main agent issues a tool call for it.

So typing `claude` launches the harness, and that act inherently instantiates the default agent that handles the session: **harness** names the engine and process; **agent** names the particular loop instance and configuration currently running inside it. At startup, those two come into existence together.

7 The Harness Landscape in 2026

Section 1 explains what a harness is, and coding-agent platforms lists common platforms as a starting point. This section is the wider map: which harnesses exist as of August 2026, how they group, and how to choose among them.

> ⚠ A fast-moving, opinionated snapshot
>
> This section condenses a research summary compiled in August 2026 (issue #95), which drew on vendor announcements, project repositories, and community discussion. Harness releases arrive weekly, default models changed several times during 2026, and the corporate landscape reshuffled more than once, so treat every named version, figure, and ownership claim as dated to that month. Check the linked project pages before acting on any of it.

The organizing idea: agent = model + harness

The consensus across the comparisons surveyed is that the frontier models have converged on the standard coding benchmarks, so the harness layers that Section 1 describes now do most of the differentiating. The commonly cited puzzle is that the same Claude model performs noticeably better inside Claude Code than inside a model-agnostic harness, and the answer offered is repeatedly the same word: the harness.

Two consequences follow for reading the rest of this section:

- **Benchmarks are saturating.** Leading models cluster within about a point on SWE-bench Verified, which is widely regarded as near its ceiling and exposed to training-data contamination. Harder successors (SWE-bench Pro, Terminal-Bench 2.x) show much lower absolute scores and separate tools better, but their figures are vendor-reported and version-specific.
- **Your own tasks are the benchmark that matters.** A day of real work on a repository you ship reveals more than any leaderboard, which is why Table 4 ends in a pilot rather than a ranking.

Terminal-first harnesses

Table 2: Terminal-first coding harnesses

Harness	License	Character (August 2026)
Claude Code (Anthropic)	Proprietary	The deepest programmable harness: hooks, skills, plugins, subagents, MCP, and multi-agent workflows. Widely rated first on code quality, and consistently the heaviest token spender, with real volume starting on the higher subscription tiers.
OpenAI Codex	Apache 2.0 CLI; proprietary cloud	The reach leader: terminal, cloud, IDE, ChatGPT app, and browser surfaces. Strong on terminal benchmarks, sandboxed at the OS level, markedly more token-efficient than Claude Code, and now carrying skills, plugins, subagents, and a small hook system.
OpenCode	MIT	The most-starred open harness: model-agnostic across many providers including local models, client-server architecture with a headless server mode, custom agents, MCP, and `AGENTS.md`. Claude works only through an API key, not a subscription login.
Aider	Apache 2.0	The git-native pioneer: maps the repository and commits each edit. Reported to be in maintenance mode since early 2026, still respected for auditability and local-model flexibility.

Harness	License	Character (August 2026)
Goose (Block, now Linux Foundation)	Apache 2.0	Rust terminal agent with vendor-neutral governance, many model providers and MCP extensions, parallel subagents, and Agent Client Protocol support. Local-first with no hosted option.
Gemini CLI (Google)	Apache 2.0	Being folded into Antigravity CLI; issue #95 records free individual access ending on 2026-06-18, with enterprise Code Assist licenses unaffected.
Amp	Proprietary	Deliberately optimizes for the best outcome regardless of token cost: multi-model routing, parallel subagents, and an "Oracle" deep-reasoning mode. Spun out of Sourcegraph.

Editor-integrated harnesses

Table 3: Editor-integrated coding harnesses

Harness	License	Character (August 2026)
Cursor	Proprietary	The in-editor leader: a VS Code fork with a deep agent loop, an in-house model tuned for fast agentic editing, and rules, MCP, hooks, skills, plugins, and subagents. Reads `.cursor`, `.claude/agents`, and `.codex/agents` configuration, per issue #95.

Harness	License	Character (August 2026)
GitHub Copilot	Proprietary	The most widely distributed option, multi-model, with a cloud agent that turns an issue into a pull request in an ephemeral Actions environment; framed by GitHub for low-to-medium complexity tasks in well-tested codebases.
Cline	Apache 2.0	The open-source VS Code standard: a plan-then-act workflow with human approval at each step, bring-your-own-key across many providers including local Ollama, and a Kanban surface for orchestrating parallel agents. The usual pick for audit-sensitive environments.
Google Antigravity	Proprietary	Google's agent-first development platform, built around an agent manager surface and now the center of its coding strategy; early reception focused on rate limits.
Devin Desktop (Cognition)	Proprietary	The former Windsurf editor, rebranded in June 2026 with a local Devin agent, an agent command center, and Agent Client Protocol support.

Continue, Tabnine, Kilo Code, Zed	Mixed	Specialists: Continue adds an open-source agent to an existing editor; Tabnine is the on-premises, air-gapped enterprise pick; Kilo Code carries on the Roo Code lineage after that project's 2026 shutdown; Zed is an open-source AI-native editor.

Autonomous and cloud harnesses

- **Devin** (Cognition) is the archetype: delegate a task, and parallel managed instances each run in an isolated cloud machine and open pull requests. Early trials were poor, and the product has matured since.
- **OpenHands** (MIT) is the open-source autonomous leader, running headless in CI to browse, edit, test, and retry.
- **The cloud tiers of interactive tools** (Copilot's cloud agent, Codex Cloud, Cursor's cloud agents) share one design assumption: the bottleneck is no longer what one agent can do, but how many you can direct and review at once. When orchestration helps covers when that parallelism is worth having.

The minimalist counter-trend

Against the deep-harness orthodoxy, a 2026 movement argues that frontier models already know what a coding agent is, so a harness should get out of the way. Its flagship is Pi (MIT), built by Mario Zechner and championed by Armin Ronacher: four tools (read, bash, edit, write), a system prompt plus tool definitions under a thousand tokens, no MCP, plan mode, to-do list, or subagents, and extension by asking the agent to write its own TypeScript extensions. In its author's own Terminal-Bench 2.0 run (post dated 2025-11-30, on Claude Opus 4.5) it placed eighth, which its author read as evidence that a minimal harness competes with the heavy ones.

The maximalist reply is that the extra tokens in a heavy harness encode product features (memory, scheduled tasks, subagents, plan mode, worktree support) rather than waste. Even Ronacher concedes that task queues, orchestration, and durable sessions will matter more over time, so the two camps may converge.

Open source versus proprietary

Open source, where you pay only your model provider or run local models:

- OpenCode
- Codex CLI
- Cline
- Aider
- OpenHands
- Goose
- Continue
- Gemini CLI
- Pi
- Zed

Proprietary:

- Claude Code
- Cursor
- GitHub Copilot
- Devin
- Amp
- Antigravity
- Replit Agent
- Tabnine

For raw capability the proprietary leaders still edge ahead, but the open options are competitive and win outright on cost, privacy, auditability, and model freedom. The 2026 signature of open-source use is bring-your-own-key, with two friction points worth knowing. Anthropic restricted third-party harnesses' use of flat-rate Claude subscriptions in April 2026, which pushed some of those users onto extra-usage or per-token billing (other models covers the same rule from the Claude Code side), and OpenCode lost Claude Pro and Max login access after a dispute with Anthropic.

Local-model coding

Local coding has matured to the point where the strongest open-weight models compete on the standard benchmarks. The community favorites for consumer hardware are the Qwen, Devstral, Gemma, and DeepSeek families, and the typical stack pairs OpenCode or Aider with a local OpenAI-compatible endpoint. Running coding agents offline and small local models cover the mechanics and the model choice.

Beyond coding: general agent frameworks

The general agent-framework layer settled into three tiers:

- **Graph-based orchestration:** LangGraph, the production default for stateful, auditable, human-in-the-loop workflows, and Mastra for TypeScript.

- **Role-based multi-agent:** CrewAI, and AutoGen, now merged with Semantic Kernel into the Microsoft Agent Framework.
- **Lab SDKs:** the OpenAI Agents SDK, Anthropic's Claude Agent SDK, and Google's ADK.

MCP and the Agent-to-Agent protocol moved to Linux Foundation stewardship, and `AGENTS.md` and `SKILL.md` have become cross-tool standards in practice. The practitioner warning that recurs in production surveys is that most deployed agents are short-horizon and single-agent, so the prevailing advice is to start single-agent; when orchestration helps makes the same case for this lab's own use.

Browser-use agents are a fast-growing adjacent category, and they remain structurally vulnerable to indirect prompt injection; do not give one credentials or financial access without sandboxing and human checkpoints.

Choosing

Table 4: Matching a harness to the job

If you want	Reach for
Maximum code quality and programmable depth, and can absorb the token cost	Claude Code on a high-volume plan; drop to Codex or Cursor if the bill or the task mix does not justify it
Cross-surface continuity, terminal tasks, and lower cost	Codex, the safest default on price for most teams
To stay in the editor	Cursor, or Copilot if you are GitHub-native and want issue-to-pull-request automation
Open source, self-hosting, local models, or auditability	Cline (editor, approval-gated), OpenCode (terminal, headless), or OpenHands (autonomous CI); Tabnine for air-gapped deployments
Hands-off delegation of well-scoped backlog work	Devin or the cloud agents in Codex, Cursor, and Copilot, with humans still reviewing the pull requests
Control, observability, and cheap parallel runs in a sandbox	Pi or Goose; stay with a heavier harness for long unattended runs

Then pilot two of them on a repository you actually ship. Token or usage-limit pain points toward Codex or an open bring-your-own-key harness; a need for audit trails points toward Cline or LangGraph; a need for private inference points toward OpenCode or Aider with a local model.

8 Custom Harnesses and How to Build Our Own

Section 1 defines a harness, Section 4 explains which of its layers are code and which are prose, and Section 7 maps the vendor harnesses. This section covers the layer that sits on top of those: the **custom harnesses** that individuals and small teams build for themselves, and what building one would mean for this lab (measured 2026-09-09 unless stated otherwise).

Two things people mean by "custom harness"

The projects surveyed here split into two kinds, and the distinction decides how much work a custom harness is:

- **A configuration layer on a vendor harness.** The execution engine stays Claude Code, Codex, or OpenCode, and the "harness" is a plugin: skills, hooks, subagent definitions, slash commands, and memory files that the vendor harness reads at runtime (the markdown-plus-front-matter layer of Section 4). `oh-my-claudecode`, Hoyeon, Superpowers, and GSD Core are all this kind.
- **A standalone runtime.** The tool-calling loop, permission model, and tool handlers are your own code, and the model is called through an API. `OpenHarness` and the LangChain guide are this kind.

The first kind is days of work and inherits every vendor upgrade for free. The second is months of work and gives you control over the loop itself, which is the only reason to pay for it.

Surveyed projects

Table 5 lists the projects surveyed.

Table 5: Custom harnesses surveyed for this section

Project	Kind	Runs on	License	Size (2026-09-09)
`oh-my-claudecode` (Heo 2026)	layer	Claude Code (plus CLI workers for Codex, Gemini, Antigravity, Grok, Cursor)	MIT	39.1k stars
Hoyeon (team-attention 2026)	layer	Claude Code	MIT	174 stars; last push 2026-05-21

Project	Kind	Runs on	License	Size (2026-09-09)
Superpowers (Vincent 2026)	layer	Claude Code, Codex, Cursor, OpenCode, Pi, Hermes, and eight others	MIT	283.8k stars; on the official Claude plugin marketplace
GSD Core (open-gsd 2026)	layer	Claude Code, OpenCode, Codex, Copilot, Cursor, and four others	MIT	9.3k stars
`OpenHarness` (HKUDS 2026)	runtime	any Anthropic- or OpenAI-compatible endpoint	MIT	15.7k stars; last push 2026-06-04
LangChain `create_agent` (Runkle 2026)	runtime	any LangChain model	MIT (library)	guide, not a product

`oh-my-claudecode` (OMC)

The `README` states: "Multi-agent orchestration for Claude Code. Zero learning curve." (Heo 2026). It installs as a Claude Code plugin (`/plugin marketplace add`, then `/plugin install oh-my-claudecode`) or as an npm CLI, and adds:

- execution modes: a staged `team` pipeline (`team-plan`, `team-prd`, `team-exec`, `team-verify`, `team-fix` in a loop), `autopilot` for single-agent autonomy, and `ralph` for persistence until a task verifies;
- 19 specialized subagents with a model-by-agent compatibility matrix and routing that sends simple tasks to Haiku and reasoning to Opus, which the README says "saves 30-50% on tokens" without a stated methodology;
- `/skillify`, which extracts a reusable skill from a session into `.omc/skills/` (project) or `~/.omc/skills/` (user) and auto-injects matching skills later;
- a HUD status line, cost tracking, Discord/Telegram/Slack notifications, and a `tmux`-based mode that drives other vendors' CLIs as workers.

It is the same family as the Oh My OpenCode project in Oh My OpenCode (the npm package is still named `oh-my-claude-sisyphus`), and by star count the most-used custom harness in this survey after Superpowers.

Useful to us? Partly. Its model routing and skill extraction are things `Morrison-Lab/ai-config` already does with `select-model` and `ums`, so installing OMC on top would double them. Its notification integrations and HUD are the pieces we lack, and they are separable.

Hoyeon

Hoyeon calls itself a "Requirements-first Harness — derive, verify, execute" (team-attention 2026). Its idea is that most agent failures are planning failures, so it forces a derivation chain (goal, context, decisions, requirements, tasks, execution) and gates each transition twice: a CLI validates the structure of `requirements.md` and `plan.json` before any model reads them, and a reviewer agent checks for scope drift and blind spots. The commands are `/specify` (a Socratic interview), `/blueprint` (contract-first planning into a task dependency graph), `/execute` (parallel workers plus independent verification against sub-requirements), `/bugfix`, `/council`, and `/tribunal` (a three-agent adversarial review: risk, value, feasibility). It ships 22 agents and 29 skills and installs with `claude plugin add team-attention/hoyeon` plus an npm CLI.

Useful to us? The design principle is exactly the lab's own "deterministic tools over model judgment" rule (customizing an agent): a CLI that rejects malformed plans before an LLM sees them is the kind of instrument our hooks already are for pushes and merges. The requirements interview is heavier than most of our tasks need, and it is Claude-Code-only.

Superpowers

Superpowers is Jesse Vincent's "agentic skills framework & software development methodology" (Vincent 2026). It is the purest example of the layer kind: a set of skills and one bootstrap that makes them mandatory. The `using-superpowers` meta-skill injects itself at session start so the agent checks for a relevant skill before any task, and the skills encode a workflow: brainstorming, writing and executing plans, test-driven development, systematic debugging, git worktrees, requesting and receiving code review, and subagent-driven development with a two-stage review (spec compliance first, then code quality). It installs from the official Claude plugin marketplace (`/plugin install superpowers@claude-plugins-official`) and from Codex's `/plugins`, and supports fourteen harnesses. Its own README records the cost of the layer approach: Hermes has no post-compaction hook, so a long session that compacts loses the bootstrap.

Useful to us? Yes, and the lab's maintainer already runs it alongside `ai-config` in Claude Code. The overlap is real (both have a brainstorming skill, both have a code-review workflow), so treat it as a reference implementation of skill design rather than a second source of truth.

GSD Core

GSD ("Git. Ship. Done.") describes itself as a "context-engineering and spec-driven development framework that drives AI coding agents through a disciplined phase loop" (open-gsd 2026). Its target is **context rot**, which it defines as the quality degradation that accumulates as an agent fills its context window. The remedy is a five-step loop per milestone:

1. discuss (capture decisions);
2. plan (research and decompose in a subagent);
3. execute (parallel executors, each starting with a fresh context);
4. verify (walk through the built work before declaring done);
5. ship (open the PR, archive the phase).

State lives in `STATE.md` and `CONTEXT.md` so it survives session boundaries. It installs with `npx @opengsd/gsd-core@latest`, which asks which of nine runtimes to target.

Useful to us? The fresh-context-executor pattern is the same one when orchestration helps recommends, and `ai-config`'s `compress-session` and `handoff` skills play the role of `STATE.md`. GSD's contribution is having made the loop the default rather than an option.

`OpenHarness`

`OpenHarness`, from the HKUDS group at the University of Hong Kong, is an "Open Agent Harness with a Built-in Personal Agent" (HKUDS 2026) and the one standalone runtime in this survey that targets coding-agent use. It is Python with a React terminal UI, installs from a shell script in the repository (the command is `oh`), and rebuilds the whole stack from Section 1: an agent loop with streaming tool use and retries, a registry of more than forty tools, Markdown skills loaded on demand, plugins that follow Claude Code's conventions (the README reports testing against twelve official plugins), three permission modes (ask, auto, plan) plus path and command rules, lifecycle hooks, MCP, persistent memory, background tasks, and subagent coordination. It talks to Anthropic, OpenAI, Copilot, Codex, Kimi, GLM, MiniMax, and NVIDIA endpoints.

Useful to us? As an escape hatch. Because it reads Claude Code skills and plugins, `ai-config` would largely carry over, so it is the cheapest way to run our configuration on a model or endpoint Claude Code cannot use (other models covers the subscription restrictions that motivate this). It is not a reason to leave Claude Code while Claude Code does what we need.

The LangChain guide

Sydney Runkle's "How to build a custom agent harness" (2026-06-03) (Runkle 2026) is the runtime kind described from the library side. It defines the harness as "the scaffolding around the model that connects it to the real world," starts from `create_agent` with three inputs (a

model, tools, a system prompt), and adds everything else as **middleware** at fixed points in the loop (before and after each model call and tool call). Its table maps eight production needs to middleware: context overflow to summarization and context editing; memory to filesystem, memory, and skills middleware; environment actions to shell, filesystem, and code-interpreter middleware; delegation to subagent and to-do-list middleware; transient failures to retry and model-fallback middleware; policy to personal-data and human-in-the-loop middleware; steering to human-in-the-loop; and cost to call limits and prompt caching. It positions the Claude Agent SDK and its own Deep Agents as pre-assembled alternatives.

Useful to us? As a checklist. Every row of that table has a counterpart in the layer approach (a hook, a skill, a permission rule), so it is a good audit of what a harness must cover whichever kind you build.

The community thread

The r/ClaudeCode thread "Show off your own harness setups here" collects personal setups of the layer kind. Reddit blocks automated fetches, so the thread is not summarized here and is worth reading directly. One small repository of the same kind, `my-claude-code-harness` (code-yeongyu 2025) (12 stars, no license, last pushed 2025-11-28), shows the shape such personal setups take: plan-reviewer, executor, and librarian agents; hooks that inject language guidelines and run a linter; and `/planner`, `/execute`, `/commit`, and `/create-pr` commands. guidance sources lists the communities where these setups are discussed.

What the projects have in common

Across the layer-kind harnesses the same five moves recur:

- a **mandatory workflow** injected at session start (Superpowers' bootstrap, OMC's prompt triggers, GSD's loop);
- **planning before execution**, with the plan written to a file (`plan.json`, `STATE.md`, a writing-plans skill);
- **fresh-context workers** for execution, with the main session kept lean;
- **independent verification** by a different agent than the one that did the work;
- **deterministic gates** where a check can be decided without a model (Hoyeon's CLI validation, `my-claude-code-harness`'s linter hooks).

The runtime-kind projects add the loop itself, the permission model, and provider independence; nothing else in the surveyed runtimes is missing from the layer kind.

How we would build our own

`Morrison-Lab/ai-config` is already a custom harness of the layer kind, and a large one. On 2026-09-09 a local checkout held:

- 191 skills;
- 94 hook scripts, and a `hooks/hooks.json` with 55 registrations (30 `PreToolUse`, 10 `UserPromptSubmit`, and 15 `Stop`);
- 8 subagent definitions (including the `adversarial-reviewer` that gates every push);
- 131 shared prose fragments and 59 memory files;
- a plugin manifest, so Claude Code and Cursor load it as a plugin (how config reaches a machine explains how, and customizing an agent what the corpus contains).

It already implements all five common moves: the bootstrap is `CLAUDE.md` and `AGENTS.md`, planning goes through issue-first and `st`, fresh-context workers are the `Agent` and `Workflow` calls, independent verification is the adversarial-reviewer subagent, and the deterministic gates are the hooks. So "building our own" means three concrete things, in order:

1. **Keep the layer approach.** Nothing in the survey justifies a standalone runtime for us yet: every vendor upgrade to Claude Code arrives for free, and the layer is portable to Codex, OpenCode, and `OpenHarness` because all of them read the same `SKILL.md` and plugin conventions.
2. **Borrow the pieces we lack.** Superpowers' two-stage review (spec compliance, then code quality) is a sharper split than our single adversarial pass; Hoyeon's structural validation of a plan file before a model reads it is a hook we do not have; OMC's notification integrations would replace our scheduled-poll check-ins. Each is one skill or one hook, and each should land as its own `ai-config` issue and PR.
3. **Decide the runtime question by a measured need, not by ambition.** The trigger for the runtime kind is a feature the vendor loop cannot give us: a model Claude Code will not call under our subscription, a permission mode it does not offer, or a loop change (a retry policy, a cost ceiling) that hooks cannot express. When that day comes, `OpenHarness` is the cheaper route, because it already loads our plugin, and the LangChain middleware table is the checklist for anything it lacks.

9 The OpenCode Ecosystem

OpenCode (anomalyco 2026) is the open-source coding agent harness (MIT-licensed, about 206,000 GitHub stars, measured 2026-09-09) that the earlier OpenCode sections of this site build on: running it against local models (connecting OpenCode to local models), against OpenRouter (connecting OpenCode to OpenRouter), and under the Oh My OpenCode multi-agent framework (Oh My OpenCode). Around the harness itself sits a community ecosystem of plugins, clients, and agent bundles, which the OpenCode maintainers index on a single documentation page (OpenCode 2026a). This section maps that page (as of 2026-09-09) and

calls out the entries most relevant to us. Where the site already covers a project, this section points there instead of repeating it.

How the ecosystem is organized

The ecosystem page groups its entries into three lists (OpenCode 2026a):

- **Plugins** (38 entries): JavaScript or TypeScript modules that OpenCode loads at startup and that hook into its lifecycle events (tool execution, file edits, session compaction, permissions, notifications, and so on). A plugin is named in the `plugin` key of `opencode.json` as an npm package, or dropped as a file into `.opencode/plugins/` (project) or `~/.config/opencode/plugins/` (global); npm packages are fetched automatically with Bun (OpenCode 2026c).
- **Projects** (11 entries): clients and integrations built on the OpenCode server API or SDK — editor front ends, web and desktop apps, a Discord bot, an Obsidian plugin, and an extension manager.
- **Agents** (2 entries): bundles of agent definitions, prompts, and commands that reshape how OpenCode plans and executes work.

The issue that prompted this section also asked about **providers**. The ecosystem page has no provider list; model providers are configured in `opencode.json` and documented separately, and the two first-party paid options are covered under "OpenCode Zen and OpenCode Go" below.

A recurring caveat applies to almost every entry: these projects are independent, and most carry an explicit "not built by the OpenCode team and not affiliated with OpenCode" disclaimer (for example `ocx` (kdcokenny 2026a) and `opencode-worktree` (kdcokenny 2026d)). Listing on the ecosystem page is a courtesy index, not an endorsement or a security review. Every plugin runs with the same permissions as OpenCode itself, so treat adding one the way you would treat adding any unaudited dependency (see benefits and hazards).

Notable plugins

Activity figures below (stars, last push, latest release) were read from the GitHub API on 2026-09-09. Several repositories have moved since the ecosystem page was written; the current location is given where it differs.

Context and token management

- `opencode-dynamic-context-pruning` (Opencode-DCP 2026) (now under the `Opencode-DCP` organization; AGPL-3.0; about 4,200 stars; release `v3.1.15` on 2026-08-16). Reduces token spend by deduplicating repeated tool calls, purging the inputs of failed tool calls after a configurable number of turns, and letting the model compress stale conversation content into summaries. The README reports prompt-cache hit rates of roughly 85%

with the plugin versus 90% without, and says development focus has shifted to a separate tool called Sleev. *Useful to us?* Yes, for long ARDI sessions on paid metered providers; the AGPL license is irrelevant for a locally-run tool but worth knowing.
- `opencode-morph-plugin` (Morph, the vendor; MIT; 85 stars; release `v2.0.17` on 2026-09-07) and its community predecessor `opencode-morph-fast-apply` (MIT; 170 stars). Both route edits through Morph's hosted "Fast Apply" model to cut edit latency, which means a second paid API and a second party seeing your code. *Useful to us?* Not by default; the privacy cost outweighs the speed gain for research code.

Subscription and quota bridges

These plugins let OpenCode consume a chat subscription you already pay for instead of API credits. They are among the most-starred plugins on the page and also the most fragile, since they depend on undocumented OAuth flows that the subscription vendors can close at any time:

- `opencode-openai-codex-auth` (Ali 2026) (Numman Ali; about 2,200 stars; release `v4.4.0` on 2026-01-09; no push since then). Uses the ChatGPT Plus/Pro OAuth flow to expose the GPT and Codex model presets, and its README restricts it to "personal development use" with your own subscription.
- `opencode-gemini-auth` (MIT; about 1,700 stars; release `v1.4.16` on 2026-05-21). The same idea for a Gemini plan.
- `opencode-antigravity-auth` (Fabris 2026) (MIT; about 11,000 stars; **archived** on GitHub as of 2026-09-09). Authenticated against Google Antigravity's model pool with a Google account, rotating across accounts to stay under quota. Its own README states that using it "violates Google's Terms of Service" and that your account "may be suspended or permanently banned". The similar `opencode-google-antigravity-auth` (375 stars) is archived as well.

Useful to us? No. Our lab policy is to use delegation budgets through their supported routes (the `codex` CLI on the ChatGPT plan, OpenCode Go or Zen, OpenRouter credits), and a bridge whose README concedes a terms-of-service violation is a liability, not a saving. The Codex and Gemini bridges are less clearly prohibited but share the fragility.

Sandboxing, isolation, and worktrees

- `opencode-daytona` (Daytona 2026) (Daytona, the vendor; Apache-2.0; release `opencode-plugin-v0.192.1` on 2026-09-03). Runs each session in a hosted Daytona sandbox, syncs the sandbox to a local branch named `opencode/<n>`, and produces live preview links when a server starts. Requires a Daytona account and API key. *Useful to us?* Maybe, for untrusted-code experiments; otherwise the firewall and container setups in this chapter's Firewall and Network Configuration section cover the same ground.
- `opencode-devcontainers` (MIT; 222 stars; release `v0.5.1` on 2026-08-24; pushed 2026-09-08). Runs several devcontainer instances at once, one per branch, with auto-assigned ports. *Useful to us?* Yes, for repositories that already ship a `.devcontainer/`.

- opencode-worktree (kdcokenny 2026d) (MIT; 726 stars; no tagged release; pushed 2026-08-17). Adds `worktree_create` and `worktree_delete` tools that create an isolated git worktree under `~/.local/share/opencode/worktree/`, sync configured files into it, and open a terminal with OpenCode running there. *Useful to us?* Yes; it is the OpenCode counterpart of the per-agent worktrees our Claude Code sessions use.

Orchestration and workflow bundles

- oh-my-opencode (now `oh-my-openagent`; about 68,800 stars; release `v5.0.0-beta.51` on 2026-09-09). Covered in Oh My OpenCode.
- opencode-workspace (kdcokenny 2026c) (MIT; 586 stars; no tagged release). A one-install bundle of 16 components: four plugins (workspace management, async delegation, notifications, git isolation), two npm plugins, three MCP servers (documentation, web search, code search), four specialist agents (researcher, coder, scribe, reviewer), four skill modules, and one command interface. Its sibling opencode-background-agents (kdcokenny 2026b) (MIT; 382 stars) provides `delegate()`, `delegation_read()`, and `delegation_list()` tools whose results persist to disk under `~/.local/share/opencode/delegations/`, so they survive context compaction; only read-only sub-agents may run in the background, because background sessions sit outside OpenCode's undo and branching system. *Useful to us?* Worth a trial as a lighter alternative to Oh My OpenCode; the disk-persisted delegation results address the same loss-on-compaction problem our lab notebook convention exists for.
- micode (MIT; 483 stars) and octto (MIT; 493 stars), from the same author: a brainstorm-plan-implement workflow with session continuity, and a browser UI that turns an agent's clarifying questions into multi-question forms. *Useful to us?* Possibly `octto`, which does for OpenCode what `AskUserQuestion` does in Claude Code.
- opencode-conductor (Apache-2.0; 129 stars; last push 2026-03-02) ports the Context, Spec, Plan, Implement lifecycle described in Conductor extension to OpenCode. *Useful to us?* Only if that lifecycle is adopted; the port looks dormant.
- plannotator (Apache-2.0; about 8,600 stars; release `v0.27.12` on 2026-09-03) is a visual plan-and-diff annotation tool with an OpenCode plugin; it is agent-agnostic and also targets Claude Code. *Useful to us?* Yes, for reviewing agent plans before implementation; it addresses the plan-review step directly.
- opencode-goal-plugin (MIT; 252 stars; release `v0.10.0` on 2026-09-07) adds a session-scoped `/goal` that keeps an objective in context and auto-continues until "evidence-gated completion". *Useful to us?* Maybe, for unattended ARDI-style loops.

Safety and observability

- opencode-vibeguard (inkdust2021 2026) (MIT; 198 stars; no tagged release; last push 2026-03-01). Replaces configured secrets and personal data with placeholders of the form `__VG_<CATEGORY>_<hash>__` before each request leaves for the model provider, restores them locally when output completes, and restores them again before tool execution so shell commands still work. *Useful to us?* Yes in principle, for work touching participant

data, but the redaction list is hand-configured and the project has been quiet for six months; test it before relying on it.

- `opencode-shell-strategy` (MIT; 136 stars) and `opencode-pty` (MIT; 571 stars) tackle the same hang: an agent running a TTY-dependent command with no terminal attached. The first injects instructions to use non-interactive flags; the second gives the agent a real pseudo-terminal it can write to and read from. *Useful to us?* Yes; the hang they fix is one we hit regularly with `R` and `quarto preview`.
- `opencode-sentry-monitor` (53 stars), `opencode-wakatime` (198 stars), and `opencode-helicone-session` (16 stars) send traces or usage to Sentry, Wakatime, and Helicone respectively. *Useful to us?* No; we do not run those services.

Notifications and quality of life

Four plugins do desktop notifications (`opencode-notifier`, 810 stars, is the most-used; `opencode-notify`, `opencode-notificator`), and the rest of the list is small conveniences: `opencode-md-table-formatter` cleans up model-generated Markdown tables, `opencode-zellij-namer` names Zellij sessions, `opencode-supermemory` (about 1,600 stars) adds cross-session memory through the hosted Supermemory service, and `opencode-scheduler` runs recurring jobs through `launchd` or `systemd`. Vendor plugins from JFrog, Firecrawl, and Tavily wrap their own CLIs. *Useful to us?* A notifier, yes; the rest only with a specific need.

Notable clients and integrations

- `CodeNomad` (Neural Nomads AI 2026) (Neural Nomads AI; MIT; about 2,600 stars; release `v0.19.0` on 2026-08-24; pushed 2026-09-09). Describes itself as "The AI Coding Cockpit for OpenCode": a SolidJS front end with a Node.js server that wraps an OpenCode CLI already on your `PATH`. It ships as Electron and Tauri desktop builds for macOS, Windows (x64 and ARM64), and Linux, and as a password-protected local web server (`npx @neuralnomads/codenomad --password <password> --launch`) for remote or browser access. Features include multi-instance workspaces, session management, git worktrees, voice input, a file browser, a command palette, and "SideCars" that embed local web tools as tabs. Despite the ecosystem page's "Desktop, Web, Mobile and Remote" description, the README documents no native mobile app; mobile access is through the web server. *Useful to us?* Yes, as the most complete graphical front end for OpenCode and a plausible answer to "I want several OpenCode sessions side by side without `tmux`".
- `OpenChamber` (OpenChamber 2026) (now under its own organization; MIT; about 9,700 stars; release `v1.22.2` on 2026-09-05). A workspace for running and reviewing agent work on desktop, web, VS Code, iOS, and Android, with session goals, a "multi-run" mode that runs the same task across up to five models, a guided changes walkthrough for large diffs, scheduled tasks, and an end-to-end encrypted "Private Relay" for remote connections. Like CodeNomad it uses OpenCode as the engine and is not affiliated with the OpenCode team. *Useful to us?* Yes; the multi-model run is a direct fit for our habit of getting a second model's review.

- OpenWork (different-ai 2026) (different-ai; about 23,400 stars; release `v0.18.44` on 2026-09-09). Positions itself as "an open-source alternative to Claude Cowork and Codex" for macOS, Windows, and Linux, sharing skills, MCP servers, and connected services across tools and machines. Code outside `ee/` is MIT; the organizational control plane under `ee/` uses a separate license that is free for up to five users. *Useful to us?* Compare against the collaborative workspaces in collaborative workspaces; the five-user free tier fits a lab, and the open core makes it auditable.
- `opencode.nvim` (Dyke 2026) (Nick van Dyke; MIT; about 3,800 stars; release `v1.0.0` on 2026-08-20) connects Neovim to a running OpenCode server (`opencode --port`), injects editor context (cursor, selection, diagnostics) into prompts, surfaces OpenCode's server-sent events as Neovim autocommands, and reloads buffers when the agent edits files. A second, unrelated `sudo-tee/opencode.nvim` (Apache-2.0; 937 stars) is a full Neovim front end rather than a bridge. *Useful to us?* For the Neovim users in the lab, yes.
- `kimaki` (remorses 2026) (MIT; about 1,400 stars; release `kimaki@0.27.0` on 2026-09-01). A Discord bot in which each project is a channel and each session a thread; it queues messages, forks sessions, transcribes voice messages, shows diffs, and maps Discord roles to permission controls. *Useful to us?* No for now; our coordination runs through GitHub, not Discord.
- `portal` (hosenur 2026) (MIT; 798 stars; last push 2026-05-12) is a mobile-first web UI meant to be reached over Tailscale. `OpenCode-Obsidian` (MIT; about 1,100 stars) embeds OpenCode in Obsidian's sidebar. *Useful to us?* Niche; CodeNomad and OpenChamber cover the remote-access case with more activity.
- `ocx` (kdcokenny 2026a) (MIT; 942 stars; release `v2.0.15` on 2026-08-17). An extension manager with portable, isolated profiles: `ocx profile add` installs a profile from a registry, `ocx oc -p <name>` launches OpenCode with it, and components are copied into `.opencode/` (the shadcn model) rather than hidden in dependencies, with SHA verification of what is installed. *Useful to us?* Yes, if we standardize an OpenCode profile for the lab; it is the closest thing to the plugin marketplace we use for Claude Code (plugins deep dive).
- `ai-sdk-provider-opencode-sdk` (MIT; 116 stars) exposes OpenCode's configured providers to the Vercel AI SDK, and `opencode-plugin-template` (archived) was the scaffold for writing plugins. *Useful to us?* Only when building on the SDK.

Agent bundles

- `OpenAgentsControl` (listed as `opencode-agents`; MIT; about 4,800 stars; release `v0.7.1` on 2026-01-30) is a plan-first framework with approval-gated execution and built-in test, review, and validation steps.
- `agentic` (MIT; 638 stars; last push 2025-09-02) is a context-engineering toolkit that appears dormant.

Useful to us? `OpenAgentsControl` overlaps heavily with Oh My OpenCode (Oh My OpenCode) and `opencode-workspace`; pick one such harness rather than layering them.

OpenCode Zen and OpenCode Go

The ecosystem page does not describe them, but two first-party paid services sit alongside the community projects:

- **OpenCode Zen** (OpenCode 2026d) is the maintainers' own model gateway: a curated set of models tested against coding-agent workloads, billed pay-as-you-go per million tokens from a prepaid balance (with optional auto-reload, by default $20 whenever the balance drops below $5). As of 2026-09-09 it lists six free models under limited-time trials, including `Big Pickle`, `MiMo-V2.5 Free`, and `Nemotron 3 Ultra Free`, beside paid Claude, GPT, Gemini, Grok, Qwen, DeepSeek, Kimi, and GLM models.
- **OpenCode Go** (OpenCode 2026b) is a $10-per-month subscription to 30-plus open-weight coding models (Qwen, DeepSeek, Kimi, GLM, MiMo, Grok, and others) with usage caps expressed in dollar value: $12 per five hours, $30 per week, $60 per month. When a cap is hit, an optional "Use balance" setting falls back to Zen credits. Only one member per workspace can hold the subscription.

Useful to us? Yes; OpenCode Go is already one of the lab's delegation budgets, and the Zen free tier is a zero-cost way to try a new open-weight model before routing work to it (compare the OpenRouter `:free` models in OpenRouter).

Summary

Need	Start with	Also see
Graphical or remote front end	`CodeNomad`, `OpenChamber`	`portal`, `OpenWork`
Multi-agent orchestration	Oh My OpenCode (Oh My OpenCode)	`opencode-workspace`, `OpenAgentsControl`
Cheaper tokens	OpenCode Go, Zen free models	`opencode-dynamic-context-pruning`
Isolation per task	`opencode-worktree`, `opencode-devcontainers`	`opencode-daytona`
Redaction of sensitive data	`opencode-vibeguard`	(none)
Plan review before implementation	`plannotator`, `octto`	`opencode-conductor`
Shareable lab configuration	`ocx`	(none)

Skip the subscription-bridge plugins.

10 VoltAgent: TypeScript Agent Framework

VoltAgent is an open-source TypeScript framework for building LLM agents, paired with a commercial console called VoltOps that adds observability, deployment, evals, and prompt management (VoltAgent 2026g). The GitHub repository reports about 10,600 stars, 1,100 forks, and an MIT license, with the latest package releases published on 2026-08-27 (measured 2026-09-09). The project started in April 2025, so it is young but actively maintained.

How agents are expressed

An agent is a TypeScript object built from a name, a set of instructions, and a model (VoltAgent 2026a):

```
const agent = new Agent({
  name: "my-agent",
  instructions: "A helpful assistant that can check weather",
  model: openai("gpt-4o-mini"),
  tools: [weatherTool],
  memory,
});
```

The `model` field takes either a model object from the Vercel AI SDK (the `ai-sdk` packages) or a `provider/model` string that VoltAgent resolves for you (VoltAgent 2026a). Because the model layer is the AI SDK, swapping between OpenAI, Anthropic, Google, and other providers is a configuration change rather than a rewrite (VoltAgent 2026g). The agent exposes `generateText()` and `streamText()` methods, and both accept an `output` schema for structured results. The default step limit is five model calls per turn (VoltAgent 2026a).

Other building blocks follow the same pattern:

- **Tools** are functions with a Zod parameter schema, plus lifecycle hooks and cancellation support. MCP servers can be attached as tool sources "without extra glue code" (VoltAgent 2026g).
- **Workflows** are chains built with `createWorkflowChain()`. Steps are added with `andThen()` (plain code), `andAgent()` (an LLM call), `andWhen()` (a conditional), `andAll()` (parallel, wait for all), and `andRace()` (parallel, first to finish wins). Input, result, and suspend/resume payloads are validated with Zod schemas, and the TypeScript type of the data flowing through the chain is inferred at each step (VoltAgent 2026f).
- **Memory** is a `Memory` object wrapping a storage adapter. Adapters exist for:
  - in-process memory
  - LibSQL/SQLite

  - PostgreSQL
  - Supabase
  - Cloudflare D1
  - a hosted "Managed Memory" on VoltOps

  Optional embedding and vector adapters add semantic recall, and a "working memory" slot holds a compact summary the agent can read and update through built-in tools (VoltAgent 2026b).

- **Supervisors and sub-agents** are ordinary agents passed in a `subAgents` array. The supervisor gains a `delegate_task` tool that takes a task description and the names of the sub-agents to call, and returns each sub-agent's text plus its token usage. A `supervisorConfig` option controls the delegation guidelines, event forwarding, error handling, and the system prompt (VoltAgent 2026d).

A `VoltAgent` instance registers the agents and workflows and serves them over HTTP through a Hono server, which is how the console and other clients reach them (VoltAgent 2026g).

Observability and hosted parts

The framework emits traces through an OpenTelemetry-based module. In development, the VoltOps console at `console.voltagent.dev` connects from the browser to the local agent process, and the documentation states that in this mode no data is sent to or stored on external servers (VoltAgent 2026e). In production, export to the hosted platform switches on when VoltOps API keys are present in the environment; Langfuse and MLflow are also listed as export targets (VoltAgent 2026c).

VoltOps itself is not open source. The README labels it "Cloud" and "Self-Hosted" (VoltAgent 2026g), but self-hosting is listed only on the Enterprise plan (VoltAgent 2026h). The plans as of 2026-09-09:

- **Developer**, free: one seat, one project, 250 traces per month, seven days of retention.
- **Core**, USD 50 per month: three seats, 50,000 traces per month, 90 days of retention.
- **Pro**, USD 250 per month: 20 seats, 250,000 traces per month, 90 days of retention, SSO.
- **Enterprise**, custom pricing: unlimited tracing and self-hosted deployment.

Overage is USD 10 per 5,000 additional traces (VoltAgent 2026h). The managed RAG "Knowledge Base" (VoltAgent 2026g) and "Managed Memory" (VoltAgent 2026b) adapters are also VoltOps features, so an agent that uses them depends on the hosted service even though the framework around them is MIT-licensed.

Useful to us?

Probably not as a replacement for what we already run. Our orchestration baseline (our orchestration baseline) is Claude Code subagents, the `Workflow` fan-out tool, and git worktrees, driven from R and Python repositories. VoltAgent's supervisor and sub-agent pattern is the same idea expressed as a TypeScript library rather than as a coding harness: the sub-agents are LLM calls with tools, not sessions with a shell and a checkout. Adopting it would mean writing our agent logic in TypeScript and hosting a Node server, which adds a language and a runtime the lab does not otherwise use.

Where it could earn a place:

- **A long-running service** rather than a coding session. A lab chatbot over the manual, or a triage bot for issue queues, is the shape VoltAgent is built for: persistent memory keyed by user and conversation, typed tools, and an HTTP server out of the box.
- **Typed, resumable workflows.** The `createWorkflowChain()` builder with Zod-validated suspend and resume payloads is more structured than what our `Workflow` tool offers for human-in-the-loop pauses.
- **Local trace inspection.** The console's localhost mode gives a trace viewer without sending data off the machine, which matters for anything touching unpublished data. The same OpenTelemetry traces can go to Langfuse or MLflow instead, so the viewer does not lock us in.

The comparison table in the comparison table covers three other orchestrators and does not include VoltAgent. On that table's dimensions, VoltAgent would read as:

- **Primary domain**: general agent infrastructure, in TypeScript.
- **License**: MIT framework, paid and closed console.
- **Maturity**: young but active.
- **Execution model**: supervisor agents delegating to sub-agents in one Node process.

11 Anthropic's Public Repositories

Anthropic publishes its code under the `anthropics` GitHub organization. The organization held 108 repositories when surveyed (measured 2026-09-09). Claude Code itself is not open source, but its public repository is still useful, and so are several others. This section inventories the repositories that matter to us, says what the two headline repositories actually contain, and ends with a verdict on each.

All counts below come from the GitHub API on the survey date (measured 2026-09-09); stars and push dates drift daily, and the license column reports what the API detected. `none` means the API found no license; for `claude-code` and `claude-agent-sdk-typescript` that is because `LICENSE.md` is a proprietary notice rather than a recognized open-source license, marked `none (proprietary)`, and for the other `none` rows there is no license file at all. `not`

`detected` means a license file exists that the API could not classify (it reports `NOASSERTION`). A single row can also hide a split: `skills` has no root license file, so the API reports none, while its README says four of its skills are source-available.

Which repositories matter to us

Forks and archived repositories together are 38 of the 108 (measured 2026-09-09): forks of infrastructure projects that Anthropic contributes patches to (`tokio`, `rayon`, `argo-cd`, `httpcore`, `orjson`, `beam`), and archived companions to research papers (`hh-rlhf`, `toy-models-of-superposition`, `sleeper-agents-paper`). Neither group concerns a lab that consumes Claude as a product. The rest sorts into six groups.

Repository	Purpose	Language	Stars	Last push	License	Verdict
`claude-code`	Issue tracker, changelog, and bundled plugins for the Claude Code CLI	Python (plugin scripts)	144.5k	2026-09-08	none (proprietary)	Track issues and read the plugins
`claude-code-action`	GitHub Action that runs Claude Code on PRs and issues	TypeScript	8.8k	2026-09-08	MIT	Already in use; read its `examples/`
`claude-code-base-action`	Read-only mirror of `base-action/` from the repository above	TypeScript	974	2026-09-08	MIT	Reference only; do not pin to it
`claude-code-security-review`	Security-focused PR review action	Python	6.2k	2026-02-11	MIT	Low priority; superseded by a review prompt

Repository	Purpose	Language	Stars	Last push	License	Verdict
`claude-agent-sdk-python`	Python SDK wrapping the Claude Code agent loop	Python	8.1k	2026-09-08	MIT	Useful for scripted agents
`claude-agent-sdk-typescript`	Issue tracker and changelog for the TypeScript agent SDK	Shell	1.7k	2026-09-08	none (proprietary)	Track issues only
`anthropic-sdk-python`	Raw Messages API client	Python	3.9k	2026-09-04	MIT	Useful for direct API calls
`sandbox-runtime`	OS-level sandbox (`srt`) for processes, MCP servers, and agents	TypeScript	5.2k	2026-09-07	Apache-2.0	Trial on Linux and macOS; alpha on Windows
`skills`	Reference Agent Skills plus the `spec/` for the skill format	Python	175.4k	2026-09-03	none	Read the spec; borrow document skills
`claude-plugins-official`	Anthropic-curated plugin marketplace	Python	36.1k	2026-09-09	Apache-2.0	Install from it; see useful plugins
`claude-plugins-community`	Read-only mirror of the community marketplace	Python	3.7k	2026-08-25	Apache-2.0	Browse before writing our own

Repository	Purpose	Language	Stars	Last push	License	Verdict
`knowledge-work-plugins`	Role-specific plugins built for Claude Cowork	Python	23.9k	2026-09-09	Apache-2.0	Templates for a lab-role plugin
`claude-cookbooks`	Notebook recipes for the Claude API	Jupyter Notebook	52.6k	2026-09-03	MIT	Reference when scripting the API
`courses`	Five API and prompting courses	Jupyter Notebook	22.8k	2026-08-28	not detected	Introductory material
`prompt-eng-interactive-tutorial`	Nine-chapter prompting tutorial	Jupyter Notebook	38.1k	2026-08-28	none	Introductory material; dated model names
`claude-quickstarts`	Starter apps you can deploy (support agent, computer use)	TypeScript	17.6k	2026-09-04	MIT	Skim only
`devcontainer-features`	Dev Container feature that installs the CLI	Shell	302	2025-12-16	MIT	Useful for a GitHub Codespaces setup
`claude-code-monitoring-guide`	Telemetry and cost-tracking guide	Markdown	369	2025-07-29	none	Read once if we meter usage

Repository	Purpose	Language	Stars	Last push	License	Verdict
`claude-ai-mcp`	Issue tracker for MCP inside claude.ai	Markdown	467	2026-06-08	not detected	Search before filing an MCP bug

The six groups, and what each is for:

- **Product trackers.** `claude-code`, `claude-agent-sdk-typescript`, and `claude-ai-mcp` exist mainly so users can file issues against a closed product.
- **Automation.** `claude-code-action`, its mirror, and `claude-code-security-review` run Claude inside GitHub Actions.
- **Programmatic access.** The `anthropic-sdk-*` clients (Python, TypeScript, Go, Java, Ruby, C#, PHP) speak the raw API; `claude-agent-sdk-python` wraps the whole Claude Code agent loop and bundles the CLI, and the TypeScript package does the same from npm, though its repository here is only a tracker.
- **Extensions.** `skills`, the two plugin marketplaces, and `knowledge-work-plugins` hold the skills and plugins the harness loads.
- **Infrastructure.** `sandbox-runtime`, `devcontainer-features`, and `claude-code-monitoring-guide` are about where the CLI runs and what it costs.
- **Learning material.** `claude-cookbooks`, `courses`, `prompt-eng-interactive-tutorial`, and `claude-quickstarts` teach the API rather than the CLI.

Two repositories one might look for are not in the organization. There is no `mcp` repository, because the Model Context Protocol lives under its own `modelcontextprotocol` organization, and the former `dxt` repository (Desktop Extensions) now redirects to `modelcontextprotocol/mcpb`, the renamed MCP Bundles format (measured 2026-09-09).

What the public `claude-code` repository actually contains

The repository named after Claude Code, second in the table only to `skills` by stars, does not contain Claude Code. The tree at `main` has 333 paths and no `src/` directory (measured 2026-09-09) (Anthropic 2026e). `LICENSE.md` is a one-line proprietary notice pointing at Anthropic's commercial terms, which is why the API reports no license. The CLI itself ships as a compiled npm package (and, since the README marked npm installation deprecated, as a native installer, a Homebrew cask, and a `winget` package; read 2026-09-09) (Anthropic 2026e), and this repository is the public face of that closed product.

The repository holds five things the lab can use:

- **`CHANGELOG.md`** is the only authoritative per-release changelog. It is the file to read when a behavior changes between versions, and its entries are detailed enough to diagnose regressions (the entry for 2.1.266, for example, names the gateway environment variable, `CLAUDE_CODE_USE_GATEWAY`, that 2.1.265 had started honoring on its own, and says no configuration change is needed; read 2026-09-09) (Anthropic 2026f).
- **`plugins/`** holds thirteen first-party plugins and a `.claude-plugin/marketplace.json` that publishes them as the `claude-code-plugins` marketplace (Anthropic 2026o). Several map directly onto lab workflows: `code-review` (five parallel review agents with confidence scoring), `pr-review-toolkit` (six specialist review agents, including a `silent-failure-hunter`), `commit-commands`, `hookify` (generates hooks from observed misbehavior), `ralph-wiggum` (a `Stop`-hook loop that keeps re-running one task), and `security-guidance` (a `PreToolUse` hook watching nine patterns). `plugin-dev` is the toolkit for writing more.
- **`examples/`** holds the reference configurations that the documentation describes in prose: `settings/` (strict, lax, and bash-sandbox profiles), `hooks/` (a Bash command validation hook), `mdm/` (managed settings for macOS and Windows fleets), and `gateway/` (AWS and GCP gateway setups).
- **`.devcontainer/`** is the container Anthropic uses for its own sandboxed sessions, including `init-firewall.sh`, the allowlist firewall that firewall configuration discusses.
- **`.github/workflows/`** is a live example of running the action against a very large issue tracker: `claude.yml` (the `@claude` agent), `claude-issue-triage.yml`, `claude-dedupe-issues.yml`, and `auto-close-duplicates.yml`, with the TypeScript behind them in `scripts/`. The repository had 12,563 open issues (measured 2026-09-09), which is why that automation exists.

So the honest description is "issue tracker plus plugins plus example configs". The harness internals are described from the outside in Section 4; nothing in this repository lets you read them.

`claude-code-action`: modes and what ships with it

`claude-code-action` is genuinely open source (MIT), and the whole action is readable: `src/` holds the entry points, the GitHub client, an in-process MCP server for file operations, and the two execution modes; `test/` has a test file per concern (comment sanitizing, branch validation, permissions, public-comment redaction, SSH signing) (Anthropic 2026g).

Claude Code Action review already explains that review is a prompt, not a separate action. The general form of that observation is the mode detector in `src/modes/detector.ts`, documented in `docs/experimental.md` (Anthropic 2026t):

1. If the workflow supplies a `prompt` input, the action runs in **agent mode**: it executes the prompt directly on whatever event fired, which is how scheduled maintenance, issue triage, and one-shot review work.

2. If there is no `prompt` but the event carries an `@claude` mention, an assignment, or the trigger label, it runs in **tag mode**: it posts a tracking comment with progress checkboxes and runs an open-ended session that can push code.
3. If neither, it does nothing.

The `track_progress` input forces tag-mode tracking comments onto `pull_request` and `issues` events that would otherwise run in agent mode. Everything else about behavior is set through `claude_args`, which passes flags straight to the CLI (`--max-turns`, `--system-prompt`, and so on), and through `plugins`, which installs named marketplace plugins before the run.

Three other things in the repository are easy to miss:

- **`examples/`** has eleven copy-paste workflows (measured 2026-09-09): the three review variants, plus `claude.yml` (the general `@claude` mention workflow), `ci-failure-auto-fix.yml`, `test-failure-analysis.yml`, `issue-deduplication.yml`, `issue-triage.yml`, `manual-code-analysis.yml`, `agent-approval-check.yml`, and `claude-wif.yml`, the workload-identity-federation variant that avoids storing a long-lived API key.
- **`agent-approval-check/`** is a second, smaller action that gates a workflow on whether the actor is a recognized agent identity, with an example identities file.
- **Authentication inputs** cover a direct API key, a Claude Code OAuth token, federation (`anthropic_federation_rule_id` plus organization, workspace, and service-account IDs), and OIDC to Bedrock, Vertex AI, or Microsoft Foundry.

The `allowed_non_write_users` input carries its own warning in `action.yml`: letting users without write access trigger the action exposes the run to prompt injection, and the secret scrubbing it performs is best-effort. `claude-code-security-review` says the same of itself in its README and recommends requiring approval for fork PRs (Anthropic 2026h). Both are the upstream statement of the caution in benefits and hazards.

The extension and SDK repositories, briefly

`skills` is the reference implementation of the Agent Skills format that Agent Skills describes. Its `spec/` directory is the format definition, `template/` is a starting skill, and `skills/` holds the examples. The README states the licensing split plainly: most skills are Apache-2.0, while the `docx`, `pdf`, `pptx`, and `xlsx` skills that power Claude's own document features are source-available rather than open source (Anthropic 2026m). Read a license header before copying one.

`claude-plugins-official` is the marketplace behind `/plugin install <name>@claude-plugins-official`, split into Anthropic-maintained `plugins/` and partner-submitted `external_plugins/` (Anthropic 2026j). Its README states a rule that applies to any marketplace we publish: a plugin's `name` is an immutable slug, since renaming it breaks every existing install, and a `renames` map exists for the unavoidable case. `claude-plugins-community`

is a nightly, read-only mirror of the community submissions that passed security scanning (Anthropic 2026i), and `knowledge-work-plugins` holds eleven role-specific bundles built for Claude Cowork (Claude Cowork) that also load in Claude Code (Anthropic 2026k).

Of the two agent SDKs, only the Python one is open source. `claude-agent-sdk-python` bundles the CLI inside the wheel and exposes `query()` and `ClaudeAgentOptions` for driving a full agent session from a script (Anthropic 2026c). `claude-agent-sdk-typescript` has the same shape as `claude-code`: a proprietary `LICENSE.md`, a changelog, examples, and an issue tracker for the npm package (Anthropic 2026d). The lower-level `anthropic-sdk-python` is the raw API client (MIT) for anyone who wants the model without the agent loop (Anthropic 2026b).

`sandbox-runtime` (`srt`) is the sandbox Claude Code uses internally, released as a beta research preview. It wraps `sandbox-exec` on macOS and `bubblewrap` on Linux with a proxy-based network allowlist, and the headline use case in its README is wrapping a local MCP server so it can read only the directories you name (Anthropic 2026l). Windows support is marked alpha, through a bundled `srt-win.exe` helper that runs the process under a dedicated local user account (measured 2026-09-09), so Windows users in the lab should treat it as experimental.

Useful to us?

Yes, selectively, and less for code than for reference.

- **Adopt now.** `claude-code-action` is already how our repositories run `@claude` and review; the `examples/` directory and `claude-wif.yml` are the upgrade path when we move off static API keys. `claude-plugins-official` is the safe default source for plugins.
- **Read, then borrow.** The `plugins/` directory in `claude-code` duplicates several things our own instruction repository does by hand — a multi-agent review command, a silent-failure hunter, a hook generator — and a `hookify`-style rule is a lighter way to encode a "never do X again" correction than a hand-written hook. `skills/spec/` is the authority when a skill fails to load.
- **Track, do not depend on.** `claude-code` and `claude-agent-sdk-typescript` are trackers. Search their issues before filing; read `CHANGELOG.md` before blaming a regression on your config. Pin workflows to `claude-code-action@v1`, never to the base-action mirror.
- **Trial on Linux and macOS.** `sandbox-runtime` could replace part of the firewall configuration in firewall configuration for macOS and Linux users; its Windows path is alpha.
- **Skip.** `claude-code-security-review` has not been pushed since February 2026 and its job is now a review prompt; the quickstarts and courses are for API programming, which is not the lab's main use of Claude.

References

Ali, Numman. 2026. *Opencode-Openai-Codex-Auth: Use Your ChatGPT Plus/Pro Subscription with OpenCode*. GitHub repository. https://github.com/numman-ali/opencode-openai-codex-auth.

anomalyco. 2026. *OpenCode: The Open Source Coding Agent*. GitHub repository. https://github.com/anomalyco/opencode.

Anthropic. 2026a. *Agent SDK Overview*. Documentation. https://code.claude.com/docs/en/agent-sdk/overview.

Anthropic. 2026b. *Anthropics/Anthropic-Sdk-Python*. Software. https://github.com/anthropics/anthropic-sdk-python.

Anthropic. 2026c. *Anthropics/Claude-Agent-Sdk-Python*. Software. https://github.com/anthropics/claude-agent-sdk-python.

Anthropic. 2026d. *Anthropics/Claude-Agent-Sdk-Typescript*. Software. https://github.com/anthropics/claude-agent-sdk-typescript.

Anthropic. 2026e. *Anthropics/Claude-Code*. Software. https://github.com/anthropics/claude-code.

Anthropic. 2026f. *Anthropics/Claude-Code: CHANGELOG.md*. Documentation. https://github.com/anthropics/claude-code/blob/main/CHANGELOG.md.

Anthropic. 2026g. *Anthropics/Claude-Code-Action*. Software. https://github.com/anthropics/claude-code-action.

Anthropic. 2026h. *Anthropics/Claude-Code-Security-Review*. Software. https://github.com/anthropics/claude-code-security-review.

Anthropic. 2026i. *Anthropics/Claude-Plugins-Community*. Software. https://github.com/anthropics/claude-plugins-community.

Anthropic. 2026j. *Anthropics/Claude-Plugins-Official*. Software. https://github.com/anthropics/claude-plugins-official.

Anthropic. 2026k. *Anthropics/Knowledge-Work-Plugins*. Software. https://github.com/anthropics/knowledge-work-plugins.

Anthropic. 2026l. *Anthropics/Sandbox-Runtime*. Software. https://github.com/anthropics/sandbox-runtime.

Anthropic. 2026m. *Anthropics/Skills*. Software. https://github.com/anthropics/skills.

Anthropic. 2026n. *Claude Code Changelog*. Documentation. https://github.com/anthropics/claude-code/blob/main/CHANGELOG.md.

Anthropic. 2026o. *Claude Code Plugins*. Documentation. https://github.com/anthropics/claude-code/blob/main/plugins/README.md.

Anthropic. 2026p. *Configure Permissions*. Documentation. https://code.claude.com/docs/en/permissions.

Anthropic. 2026q. *Create Custom Subagents*. Documentation. https://code.claude.com/docs/en/sub-agents.

Anthropic. 2026r. *Create Plugins*. Documentation. https://code.claude.com/docs/en/plugins.

Anthropic. 2026s. *Environment Variables*. Documentation. https://code.claude.com/docs/en/env-vars.

Anthropic. 2026t. *Experimental Features*. Documentation. https://github.com/anthropics/claude-code-action/blob/main/docs/experimental.md.

Anthropic. 2026u. *Explore the Context Window*. Documentation. https://code.claude.com/docs/en/context-window.

Anthropic. 2026v. *Hooks Reference*. Documentation. https://code.claude.com/docs/en/hooks.

Anthropic. 2026w. *How Claude Code Works*. Documentation. https://code.claude.com/docs/en/how-claude-code-works.

Anthropic. 2026x. *How Claude Remembers Your Project*. Documentation. https://code.claude.com/docs/en/memory.

Anthropic. 2026y. *Tools Reference*. Documentation. https://code.claude.com/docs/en/tools-reference.

anthropics/claude-code contributors. 2026a. *[BUG] V2.1.150 Adds Server-Side System Prompt Injection via Tengu_heron_brook Feature Flag*. GitHub issue #62061, anthropics/claude-code. https://github.com/anthropics/claude-code/issues/62061.

anthropics/claude-code contributors. 2026b. *[BUG] V2.1.219 Heron_brook Prompt Section Injects "Do Not Call the AgentTool Unless the User Requested It" for Opus 5 Only, Silently Overriding User-Configured Delegation Policy, with No Opt-Out*. GitHub issue #80988, anthropics/claude-code. https://github.com/anthropics/claude-code/issues/80988.

anthropics/claude-code contributors. 2026c. *Dynamic System-Prompt Sections Silently Override CLAUDE.md; No Precedence Rule, No Way to Observe It*. GitHub issue #80998, anthropics/claude-code. https://github.com/anthropics/claude-code/issues/80998.

anthropics/claude-code contributors. 2026d. *[MODEL] Model Attributes the Opus-5-Only Heron_brook Delegation Directive to the User's Own CLAUDE.md, Sending Users to Audit Config for a Rule That Isn't There*. GitHub issue #87635, anthropics/claude-code. https://github.com/anthropics/claude-code/issues/87635.

anthropics/claude-code contributors. 2026e. *Undocumented System-Prompt Directive "Do Not Call the AgentTool Unless the User Requested It" Contradicts Documented Description-Based Subagent Delegation*. GitHub issue #82456, anthropics/claude-code. https://github.com/anthropics/claude-code/issues/82456.

cnighswonger. 2026. *Heron_brook Bootstrap Channel: Disclosure Record (2026-05)*. Documentation. https://github.com/cnighswonger/claude-code-cache-fix/blob/main/docs/disclosure/heron-brook-2026-05.md.

code-yeongyu. 2025. *My-Claude-Code-Harness*. Software. https://github.com/code-yeongyu/my-claude-code-harness.

Daytona. 2026. *Daytona Integrations: OpenCode Plugin*. GitHub repository. https://github.com/daytona/integrations/tree/main/packages/opencode-plugin.

different-ai. 2026. *OpenWork: The Open-Source Alternative to Claude Cowork*. GitHub repository. https://github.com/different-ai/openwork.

Dyke, Nick van. 2026. *Opencode.nvim: Neovim Plugin for Editor-Aware Prompts*. GitHub repository. https://github.com/nickjvandyke/opencode.nvim.

Fabris, Noe. 2026. *Opencode-Antigravity-Auth (Archived)*. GitHub repository. https://github.com/NoeFabris/opencode-antigravity-auth.

Heo, Yeachan. 2026. *Oh-My-Claudecode: Multi-Agent Orchestration for Claude Code*. Software. https://github.com/yeachan-heo/oh-my-claudecode.

HKUDS. 2026. *OpenHarness: Open Agent Harness with a Built-in Personal Agent*. Software. https://github.com/HKUDS/OpenHarness.

hosenur. 2026. *Portal: Mobile-First Web UI for OpenCode*. GitHub repository. https://github.com/hosenur/portal.

inkdust2021. 2026. *Opencode-Vibeguard: Redact Secrets and PII Before LLM Calls*. GitHub repository. https://github.com/inkdust2021/opencode-vibeguard.

kdcokenny. 2026a. *Ocx: OpenCode Extension Manager with Portable, Isolated Profiles*. GitHub repository. https://github.com/kdcokenny/ocx.

kdcokenny. 2026b. *Opencode-Background-Agents: Async Delegation with Context Persistence*. GitHub repository. https://github.com/kdcokenny/opencode-background-agents.

kdcokenny. 2026c. *Opencode-Workspace: Bundled Multi-Agent Orchestration Harness*. GitHub repository. https://github.com/kdcokenny/opencode-workspace.

kdcokenny. 2026d. *Opencode-Worktree: Zero-Friction Git Worktrees for OpenCode*. GitHub repository. https://github.com/kdcokenny/opencode-worktree.

matheusmoreira. 2026. *Tell HN: Claude Code Now Allows Anthropic to Remotely Inject System Prompts*. Hacker News thread. https://news.ycombinator.com/item?id=48259288.

Neural Nomads AI. 2026. *CodeNomad: The AI Coding Cockpit for OpenCode*. GitHub repository. https://github.com/NeuralNomadsAI/CodeNomad.

OpenChamber. 2026. *OpenChamber: Agentic Development Environment for OpenCode*. GitHub repository. https://github.com/openchamber/openchamber.

OpenCode. 2026a. *OpenCode Documentation: Ecosystem*. Documentation. https://opencode.ai/docs/ecosystem/.

OpenCode. 2026b. *OpenCode Documentation: Go*. Documentation. https://opencode.ai/docs/go/.

OpenCode. 2026c. *OpenCode Documentation: Plugins*. Documentation. https://opencode.ai/docs/plugins/.

OpenCode. 2026d. *OpenCode Documentation: Zen*. Documentation. https://opencode.ai/docs/zen/.

Opencode-DCP. 2026. *Opencode-Dynamic-Context-Pruning: Optimize Token Usage by Pruning Obsolete Tool Outputs*. GitHub repository. https://github.com/Opencode-DCP/opencode-dynamic-context-pruning.

open-gsd. 2026. *GSD Core: Git. Ship. Done.* Software. https://github.com/open-gsd/gsd-core.

r/ClaudeCode. 2026. *Claude Code Has a Hardcoded Instruction Telling …* Reddit thread. https://www.reddit.com/r/ClaudeCode/comments/1v6y5q2/claude_code_has_a_hardcoded_instruction_telling/.

remorses. 2026. *Kimaki: Discord Bot to Control OpenCode Sessions.* GitHub repository. https://github.com/remorses/kimaki.

Runkle, Sydney. 2026. *How to Build a Custom Agent Harness.* Documentation. https://www.langchain.com/blog/how-to-build-a-custom-agent-harness.

team-attention. 2026. *Hoyeon: Requirements-First Harness.* Software. https://github.com/team-attention/hoyeon.

Vincent, Jesse. 2026. *Superpowers: An Agentic Skills Framework and Software Development Methodology.* Software. https://github.com/obra/superpowers.

VoltAgent. 2026a. *VoltAgent Documentation: Agents Overview.* Documentation. https://voltagent.dev/docs/agents/overview/.

VoltAgent. 2026b. *VoltAgent Documentation: Memory Overview.* Documentation. https://voltagent.dev/docs/agents/memory/overview/.

VoltAgent. 2026c. *VoltAgent Documentation: Observability Overview.* Documentation. https://voltagent.dev/docs/observability/overview/.

VoltAgent. 2026d. *VoltAgent Documentation: Sub-Agents.* Documentation. https://voltagent.dev/docs/agents/sub-agents/.

VoltAgent. 2026e. *VoltAgent Documentation: VoltOps Developer Console.* Documentation. https://voltagent.dev/docs/observability/developer-console/.

VoltAgent. 2026f. *VoltAgent Documentation: Workflows Overview.* Documentation. https://voltagent.dev/docs/workflows/overview/.

VoltAgent. 2026g. *VoltAgent: AI Agent Engineering Platform Built on an Open Source TypeScript AI Agent Framework.* Software. https://github.com/VoltAgent/voltagent.

VoltAgent. 2026h. *VoltOps Pricing.* Documentation. https://voltagent.dev/pricing/.

wtfwhs. 2026. *Tengu-Decoded: Internal Codenames, Version 2.1.169.* Software. https:

//github.com/wtfwhs/tengu-decoded/blob/main/versions/2.1.169/codenames.md.