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

Authors: Paul Barbaste, Tristan Darrigol, Germain Vu, and Tom Wiltberger

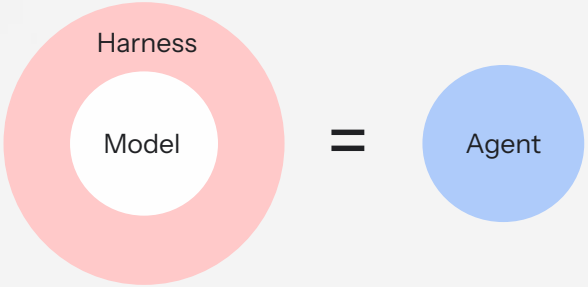
Presented by: Jacob Tjaden and Jace Chen

What is a harness?

Agent = Model + Harness

The model supplies *intelligence*—the harness turns it into *work* and makes it *usable in production*

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

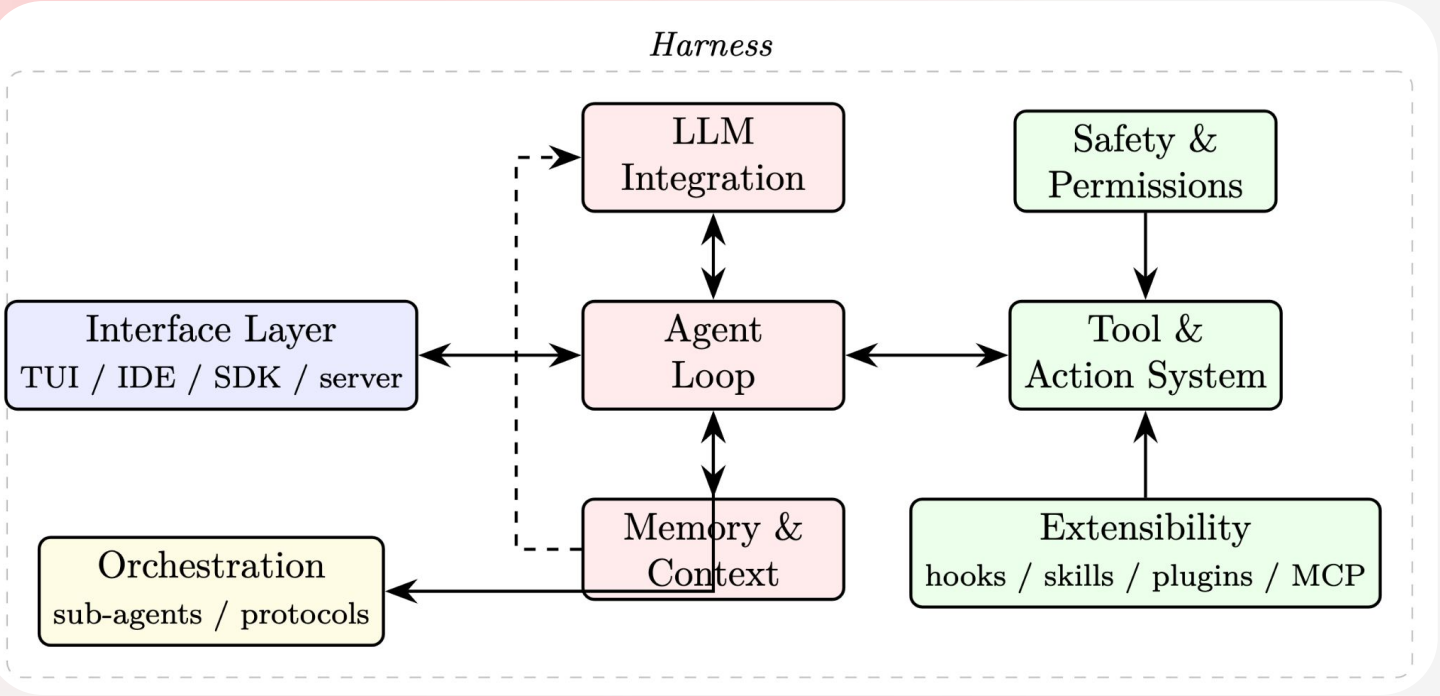


What a harness is NOT

1. Scaffold
2. Agentic framework
3. Evaluation harness
4. Orchestrator

7 subsystems of a harness

A harness has a multitude of components that add to agent capability



7 subsystems

All of these components can be improved to make the agent better

Related work

Agent = Model + Harness → When was this equation popularized?

"Every vendor now ships a harness, and the market is consolidating"

The harness layer is no longer a greenfield. It is becoming an established **ecosystem** and **economy**

Related works have explored **multi-agent collaboration**

1. MetaGPT

Assigns special roles to each agent (a product manager, architect, engineer, etc)

2. ChatDev

Organizes a role-playing pipeline

3. AutoGen

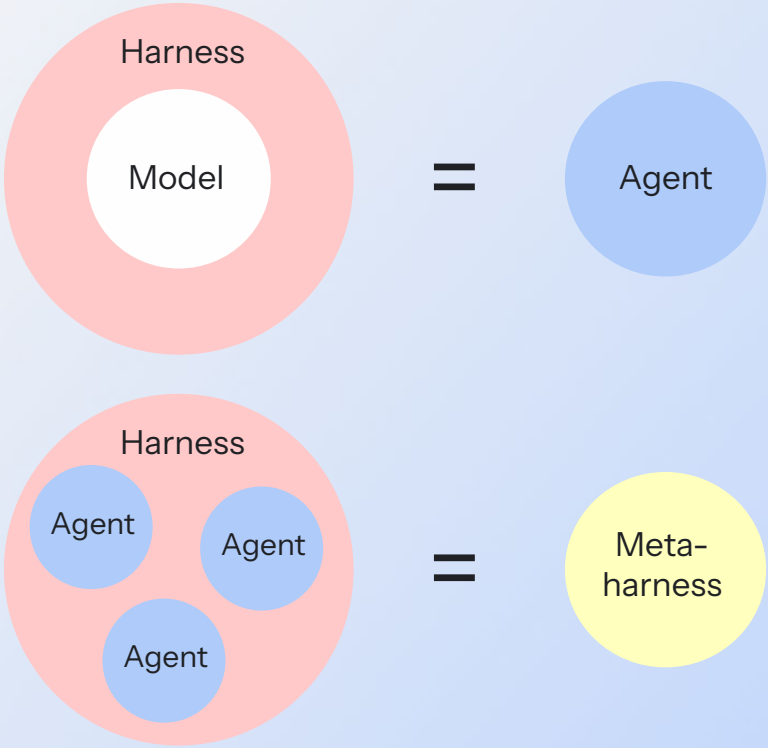
Provides a framework for multi-agent conversations

Methodology

This paper tracks 11 coding harnesses:

- Claude Code
- Codex
- Gemini CLI
- Mistral Vibe
- OpenHands
- Aider
- Mini-SWE-Agent
- Hermes
- Pi
- OpenClaw
- Omnigent*

**Omnigent is a meta-harness: it lets different agents interact with one another inside its own sandbox*



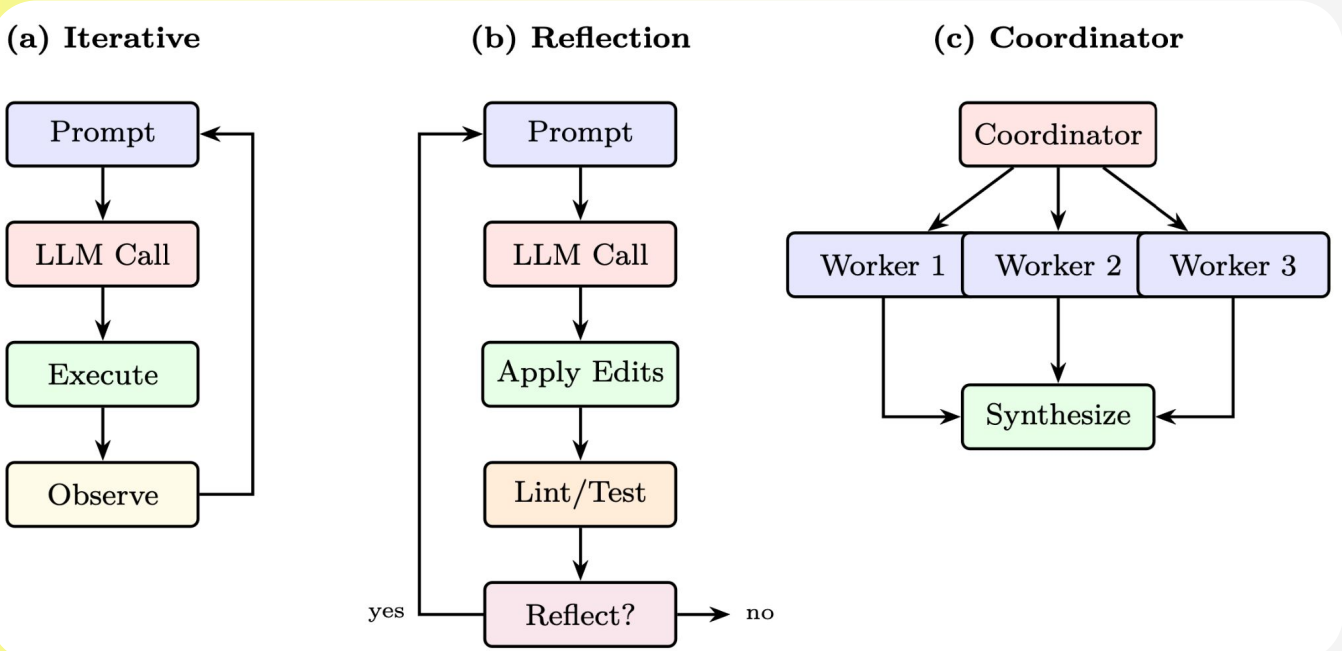
Paper's Goal: Analyze each system under the seven subsystems:

(1) Agent loop design, (2) Tools, (3) LLM integration, (4) Memory, (5) Safety, (6) Multi-agent orchestration, (7) Extensibility

Agent loops (subsystem 1)

The agent loop is the **central control flow** that alternates between LLM inference and action execution

3 loop paradigms



Iterative action-observation

- Follows cycle of (1) prompt construction, (2) LLM inference, (3) tool execution, (4) observation

Reflection-augmented

- Reflection mechanism: after each LLM response, the system applies edits and checks for lint errors. If issues are detected, a *reflected_message* re-invokes the LLM with corrective context

Coordinator-worker

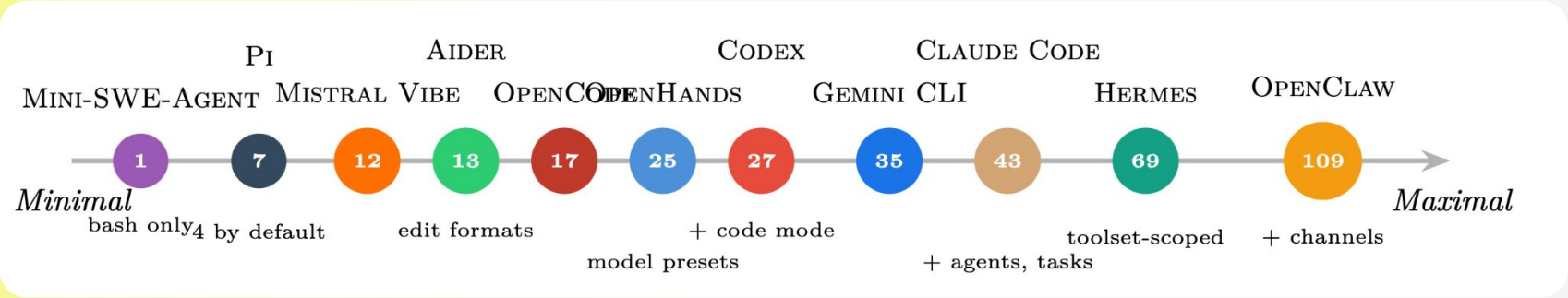
- Adds a hierarchical dispatch layer of subagents to the iterative loop. Parent agent orchestrates multiple worker agents

Tools & actions (subsystem 2)

The tool system defines what the agent can actually do: what actions it can invoke and how they execute

This system absorbs a large share of the engineering effort in every mature agent studied by this paper

Tool count spectrum



Tools range in abstraction → Mini-SWE-Agent sends everything through a **single shell call**
→ Claude Code uses **43 tools for each** declare validation, permission check, and UI render

Meta-tool: Claude Code utilizes *deferred tool loading* where it uses the *ToolSearchTool* to find tools it needs in the moment
This significantly reduces prompt size (tools are not pre-loaded)

ROADMAP

Prompts, Caching & Context Management

§7

How does the harness talk to the model?

- Provider coupling spectrum
- Prompt assembly & caching
- What the prompts actually say

§9

What happens when the conversation outgrows the window?

- Four context strategies
- Threshold compaction, seven flavors
- Persistent memory: who writes it?

§13.2

What do all eleven systems leave out?

- No agentic frameworks
- No vector RAG over code
- Why that is rational

Corpus (11 harnesses): Claude Code · Codex CLI · Gemini CLI · Mistral Vibe · OpenHands · Aider · Mini-SWE-Agent · Hermes · Pi · OpenCode · OpenClaw

PRIMER · FOUNDATIONS

The model remembers nothing: every turn resends everything

A request is not just your question — it is the whole situation, rebuilt from scratch

ONE REQUEST TO THE MODEL

System prompt
identity, rules, tone, tool-use guidance

Tool schemas
every tool's name, arguments and description (Claude Code: 43 tools)

Context files
`CLAUDE.md` / `AGENTS.md` discovered in the repository

Conversation so far
every question, answer, tool call and tool result

Your new message
the only part that is actually new

…and all of it again next turn.

THREE CONSEQUENCES

1 **It gets expensive**
The same instructions are re-read every single turn.
→ prompt caching (§7)

2 **It cannot grow forever**
Everything must fit inside one context window.
→ compaction (§9)

3 **Cross sessions memory**
The model knows only what is in this request (Statelessness).
→ context files & persistent memory (§9)

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

§7 LLM INTEGRATION (subsystem 3)

Five ways to couple a harness to its model

Tight coupling → Provider-agnostic

1 Single provider

Claude Code · Codex · Gemini CLI

Vendor SDK only. Depends on vendor features: extended thinking, cache boundaries, Responses API.

2 Provider-first + fallback

Mistral Vibe

Deep Mistral backend; a generic backend speaks Anthropic, Vertex, and OpenAI-compatible APIs.

3 Abstraction library

OpenHands · Aider · Mini-SWE-Agent

LiteLLM gives 100+ models. Aider adds per-model metadata for 350+ models.

4 Owned transports

Hermes · Pi · OpenCode

Every wire protocol hand-written (Pi: 9 protocols, 35 providers).

5 Plugin-based

OpenClaw

A plugin interface; each provider is an adapter module..

4 vs 5: both hand-built — what differs is the unit

Owned transports — the harness owns the **protocol code**; a provider's quirks are declarative data (Hermes: 5 transports, 29 profiles)

Plugin-based — the harness owns an **interface**; each provider is a self-contained adapter module (OpenClaw: 10+ adapters)

Fun fact: Harness mimicry

On a Claude subscription token, Pi presents itself as Claude Code: same prompt opening, headers, and tool-name casing.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

PRIMER → §7

Prompt caching: pay once for what never changes

1

Turn 1

The server reads all ~10,000 tokens of instructions and tool schemas, then answers.

2

Turn 2

The very same 10,000 tokens arrive again. The server recognises the identical opening and reuses the work it already did — a **cache hit**: far cheaper, far faster.

3

The catch

The match starts at the **first token** and stops at the **first difference**. It is a bookmark, not a search.

Put today's date at the top and nothing ever matches: you pay full price for 10,000 tokens, every turn, forever.

WHAT THE SERVER SEES ON TURN 2

System prompt	identical
Tool schemas	identical
Earlier turns	identical
first difference — the bookmark ends here	
New message	processed
The model's answer	generated

The design rule: stable content first, volatile content last.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

Some mechanisms for caching prompt

A cache hit needs the opening of the request to be identical to last time

Claude Code

Splits the prompt at a fixed marker

Sections that never change go before the marker, which is hashed; per-turn data goes after it.

Only the text before the first change can be reused.

```
identity · rules · tone [MARKER]
session · memory · environment
```

Codex

Labels the request with the conversation ID

Sends a label instead of hashing the text; the server stores the cache under it.

Simpler to build, but the cache serves only that conversation.

```
prompt_cache_key: "thread_a91f3c"
```

Hermes

Rewrites the prompt into one fixed form

Sorts JSON keys, sends a date instead of a clock time, freezes memory files.

Reordered keys or a ticking clock change the bytes and lose the cache.

```
{"b":2,"a":1} → {"a":1,"b":2}
14:32:08 → 2026-09-17
```

Pi

A monitoring tool - Marks cache points and measures the waste

Sets where caching starts and how long it lasts, then reports what misses cost.

A lost cache looks normal in logs; the cost only shows up on the bill.

```
[cache from here · keep 1h]
report: dollars lost per turn
```

OpenCode

Writes six providers' cache markers at once

Every provider has its own syntax for marking a cache point; it emits them all in a single request.

One prompt then caches on any provider, with no branching in the code.

```
cache_control (Anthropic) +
prompt_cache_key (OpenAI) + 4 more
```

The other six

OpenHands splits stable from per-turn exactly as Claude Code does, found independently. Aider and Mini-SWE-Agent send the provider's default markers. Mistral Vibe caches only its git-status block. OpenClaw leaves it to each adapter. Gemini CLI's API offers no request cache.

§7 LLM INTEGRATION

Each system rebuilds its prompt from different inputs

The system prompt is generated per request, not stored as one fixed text

Codex

A different prompt per model version

GPT-5.2 and GPT-5.6 get different instructions, downloaded as data from a server.

A new model can be tuned without shipping a new version of the app.

```
models.json ← /models endpoint
```

OpenCode

A different prompt per model family

Nine base prompts, picked by matching the model's name.

The same wording produces different behaviour on different model families.

```
model id "claude-…" → claude.txt
```

Mistral Vibe

A different prompt per agent role

The plan, explore and default agents each load their own Markdown file.

Sub-agents need different instructions without branching in the code.

```
prompts/plan.md · explore.md
```

Aider

A different prompt per editing mode

13 ways of writing file edits, each with its own prompt, examples and reminders.

The model must be told exactly how to format edits in that mode.

```
diff · udiff · whole · patch …
```

Pi

A prompt that shrinks with the tool list

Each enabled tool adds its own usage note; 30–40 lines in total.

Instructions for switched-off tools would waste tokens and mislead the model.

```
4 tools on → 4 usage notes
```

The other six

Three shared approaches

Claude Code · OpenHands · Hermes — one prompt for every turn; only the date, folder and memory change, and those go last so the rest stays cached.

Gemini CLI — different instructions for older vs newer models.

OpenClaw — built from the session config.

Mini-SWE-Agent — one template, filled once.

All eleven are covered: five vary their prompt on one input each; the other six share the three approaches in the last card.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

Four instructions that appear in prompt after prompt

Written by teams who never coordinated — because the models share the same habits

1

Do only what was asked

Six prompts forbid unrequested features, refactors or extra files. Models tend to over-deliver when a request is vague. Pi is the deliberate exception: it ships no such rule.

2

Put a number on reply length

Nine of eleven limit verbosity — OpenCode demands under 4 lines, Mistral Vibe ~150 words, Claude Code ≤ 25 words between tool calls.

3

Ban filler words and emoji

Mistral Vibe forbids “robust”, “seamless”, “Great!” and every emoji; OpenCode's GPT prompt also bans em dashes. These are habits of LLM writing that users notice and dislike.

4

Read a file before editing it

Required by Claude Code, Mistral Vibe and OpenCode. But OpenCode's tool description claims it will error if you skip the read, and no such check exists in the source.

Who does not: Pi has no such rule; OpenHands sets no length cap.

How a prompt makes a rule stick

Capital letters

`IMPORTANT:` and `NEVER` inline — Claude Code, Codex, OpenHands, OpenCode

Ranked rule blocks

Mistral Vibe ranks 7 levels of instruction priority, highest wins

Worked examples

Aider and Hermes show right-and-wrong pairs instead of adjectives

No prompt refuses anything

None of the eleven contains a rule of the form “if the user asks for X, refuse”. Harnesses leave that entirely to the model and the provider.

The same rule, three months apart: may the agent commit?

APRIL 2026

Every prompt that mentions git forbids autonomous commits.

Every system agreed — the strongest agreement in the study

JULY 2026

Still forbidden

Claude Code · OpenCode · Hermes · Codex (GPT-5.2 prompt)

Reversed

Mistral Vibe now teaches how to commit, with a Co-Authored-By trailer

Never forbidden

OpenHands always taught commits; gates push & PRs on request

Dropped

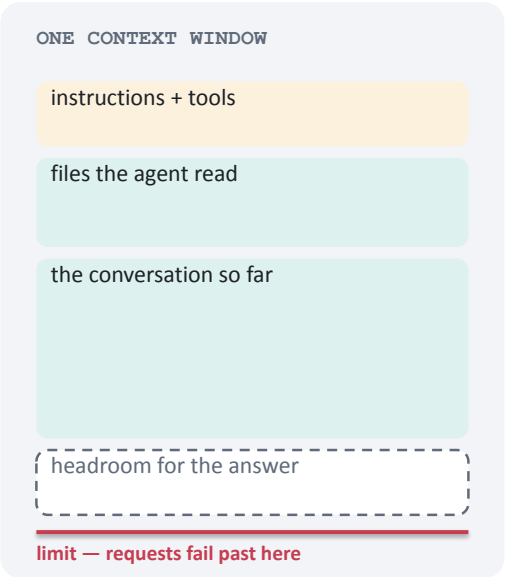
Codex GPT-5.6 prompt: the word “commit” no longer appears

Still universal: no prompt allows autonomous push, force-push or history rewriting. **The other five prompts never mention git at all.**

PRIMER → §9

When the context window fills, something has to go

The window is a hard limit: instructions, tools, files read and the whole conversation share it



WHY THIS IS HARD

1

You cannot simply delete the oldest messages

The original task, the constraints and the decisions already made all live at the top of the conversation. Drop them and the agent forgets what it is doing.

2

A summary is lossy by definition

Replacing 50 turns with a paragraph discards file paths, error messages and half-finished work the agent may still need.

3

Any rewrite of history breaks the prompt cache

Everything after the edited point must be re-read, so the turn right after compaction is the most expensive one in the session.

So every design trades **fidelity against cost**. The corpus offers four answers, and the field has converged on one.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

§9 MEMORY & CONTEXT
(subsystem 4)

Four ways to make a conversation fit again

Every card answers the same two questions: when does it fire, and what survives it

1 system

`LINEAR HISTORY`

Do nothing at all

Mini-SWE-Agent

Fires: never

Keeps: everything — until the window runs out and the run stops

Deliberate: every run stays reproducible.

2 systems

`RECURSIVE SUMMARIZATION`

Keep history under a fixed budget

Aider · OpenClaw

Fires: constantly, far below the model's limit

Keeps: the newest half word-for-word; the oldest half is summarized by a cheaper model, up to 3 times over

1 system

`PLUGGABLE CONDENSATION`

Choose the method by config

OpenHands

Fires: when configured — or on an overflow, instead of crashing

Keeps: whatever the chosen plug-in returns: nothing, a summary, or a chain of steps

Ten shipped, later cut to three.

7 systems

`THRESHOLD COMPACTION`

Summarize once the window is nearly full

Claude Code · Codex · Gemini CLI · Mistral Vibe · Hermes · Pi · OpenCode

Fires: rarely — only as the model's context limit approaches

Keeps: a recent tail word-for-word (30%, or 20K tokens, or 2 turns); everything older becomes one summary

Cards 2 and 4 both summarize — the difference is when, and how much. Aider prunes all session long to stay under its own budget; the seven stay full-size until the model's ceiling, then replace everything except a recent tail.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

§9 MEMORY & CONTEXT

Seven systems summarize near the limit — with different settings

System	When it fires	What is different about it
Claude Code	13K tokens below the effective window	Summary + boundary marker; re-attaches ≤ 5 files (50K) and skills (≤ 25K each)
Codex	Configurable token limit	Local or server-side summarization (`/responses/compact` endpoint)
Gemini CLI	50% of the model's token limit	Keeps newest 30% verbatim; per-file levels FULL / PARTIAL / SUMMARY / EXCLUDED
Mistral Vibe	Middleware threshold check each turn	Re-injects earlier user messages so goals survive; also fires on mid-turn overflow
Hermes	50% of window (64K floor)	**Compaction ends the session**: a child session links to its parent; no history lost
Pi	Window − 16,384 (keeps 20K recent)	Merges into the previous summary; Store sessions as a tree with fork / rewind
OpenCode	Input limit − reserved output (chars ÷ 4)	Forces summary into fixed sections: Objective · Details · Work State · Next Move · Files

Trend: the newer systems mix in the other strategies — merging summaries, session trees. **The remaining four** (Mini-SWE-Agent, Aider, OpenClaw, OpenHands) use the simpler methods on the previous slide.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

§9 MEMORY & CONTEXT

How agents keep notes between sessions — and get them back

Four designs, one question each: who or what is in charge of the memory

AN AGENT WRITES IT

Codex

past sessions

↓

LLM extracts memories

↓

sandboxed sub-agent consolidates into files (git-tracked)

A PERSON APPROVES IT

Gemini CLI

finished transcripts

↓

sub-agent writes patch to .inbox/

↓

user reviews via /memory

THE MODEL WRITES IT, CAPPED

Hermes · Claude Code · OpenHands

model edits Markdown file

↓

hard caps: 2,200 / 1,375 chars

↓

snapshot frozen per session

Hermes freezes the file at session start, so writes never disturb the cache.

Search instead of pasting

OpenClaw

user message arrives

↓

memory sub-agent recalls

↓

main agent replies

Searches a store each turn instead of pasting a file into the prompt.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

PRIMER → §13.2

Two things that are missing

AGENTIC FRAMEWORK

A library you import

- LangChain, LangGraph, AutoGen, CrewAI and a dozen others.
- They ship the plumbing: an agent loop, a tool registry, memory, retries, provider adapters.
- **Why reach for one?** Don't rebuild the loop; swap models freely; follow patterns others debugged.

*The paper's distinction: a framework is a library you **import**; a harness is a runtime you **work inside**.*

RAG OVER CODE

Retrieval-Augmented Generation

- Split the repository into chunks; turn each chunk into an embedding (a list of numbers capturing meaning); store them in a vector database.
- At query time, embed the question, find the nearest chunks, paste them into the prompt.
- **Why reach for it?** The repo is far bigger than the window, and it finds code by meaning rather than exact words.

Standard practice in document chatbots — and the obvious fit for a million-line codebase.

Both are standard advice for LLM applications. Across ~4M lines of production agent code, **neither one appears.**

Absence #1: no agentic frameworks

0/11

agent runtimes import a general-purpose agentic framework

Manifests inspected and source trees grepped across ~4M lines of Python, TypeScript & Rust

Searched for

LangChain · LangGraph · LlamaIndex · AutoGen · CrewAI · Pydantic AI · Genkit · Haystack · Semantic Kernel · Google ADK · Smolagents · Swarm · Agno

Gemini CLI uses neither of Google's own frameworks (Genkit, ADK).

Instead: every loop is hand-rolled in native async primitives

`Asyncio (Python)`	`sync (Python)`	`Promise / async iterator (Javascript)`	`Tokio (Rust)`
OpenHands · Mistral Vibe · Hermes	Aider · Mini-SWE-Agent	Claude Code · Gemini CLI · Pi · OpenCode · OpenClaw	Codex

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

Absence #2: no vector RAG over code

0/11

systems retrieve code with vector embeddings

Checked: Chroma, Pinecone, Weaviate, Qdrant, Milvus, FAISS, LanceDB, sqlite-vec, embedding libraries

What they use instead

rg

Search the text, like a developer would

`ripgrep`, glob and find, bundled or downloaded on first use. **All eleven** do this — Codex uses a Rust file search, OpenHands and Mini-SWE-Agent go through the shell

AST

Build a map of symbols from the syntax

Aider parses the code and ranks functions and classes by relevance, under a token budget — the corpus's only repo map

.md

Context files

`CLAUDE.md` / `AGENTS.md` / `GEMINI.md` found up the directory tree; nested ones attached just-in-time

LSP

Ask the compiler after each edit

Language servers (parser, linter, name/type checkers, etc) report errors on what was just written (OpenCode runs ~25 of them), with no stored index

Embeddings appear only for chat memory: OpenClaw's default hybrid search (sqlite-vec + BM25). Hermes deliberately stays lexical.

Why the absences are rational

Why no frameworks

1. Abstraction layers hide the prompts and responses you need to debug
2. **These failures are silent — and the agent has write access.** A corrupted prompt, a dead cache or a stale tool format shows up as odd behaviour and a bigger bill, never an error.
3. Anthropic's own 2024 guidance said to start from raw API calls rather than a framework

Why no code RAG

1. Code has deterministic structure (paths, types, parses, LSP) that similarity search can't match (i.e. Two modules each with a parse() function look alike to an embedding, but import resolution knows exactly which one a given file uses)
2. Code changes minute to minute, so pre-built embeddings are stale by construction
3. ripgrep / find / glob are already near-optimal; RAG adds compute, index upkeep and drift
4. A benchmark (CodeRAG-Bench) found retrieval helps on some tasks and not others (inconsistency)

Debuggability and prompt transparency outweigh framework reuse when the failure mode is mutating real code.

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

WRAP-UP

Takeaways

1. Prompts are built so the stable half can be cached
2. Built in systems Instructions per model/role/edit/tools
3. Shrinking a conversation before max context window is reached
4. ripgrep and Markdown files > RAG

Harness Engineering: A Source-Code Study of Eleven Systems (Barbaste et al., 2026)

Safety and permission models (subsystem 5)

Safety proceeds autonomy:
Agents cannot be given more autonomy without more advanced safety mechanisms

Example:

Mini-SWE-Agent

Scores **74%** on SWE-Bench Verified
Harness size: ~100 lines of code

Codex

Scores **69%** on SWE-Bench Verified
Harness size: Tens of thousands of lines of code

*not a fair one-to-one comparison...
but illustrates that a LOT of Codex's codebase is not invested into *task-completion logic*, but to *safety*

This is the difference between an agent used in **production** vs. an agent only used for **benchmarking**

Claude Code vs. Codex vs. Gemini CLI (overview)

These are the most dominant commercial coding agents, but **safety mechanisms** are designed differently

1. Gemini CLI

Focused on *approval modes* that control **tool execution** and **action authorization**

2. Claude Code

Compositional-prescriptive: enforced **layered review** via structured phases

3. Codex

Sandbox-emergent: safety enforced at **OS level**

Safety in Gemini CLI

**A cross-platform sandbox is an isolated virtual workspace that works across different operating systems and devices*

Overarching structure:

1. Four **approval modes**
2. Cross-platform sandbox*

plan: Strict, read-only mode

default: Interactive mode where all write actions require explicit confirmation

auto_edit: Automatically approves file edit tools while still requiring confirmation for other actions

yolo: Autonomous mode that automatically approves all tool calls and actions without prompting

Gemini CLI Approval Modes

Safety in Claude Code

Claude has an elegant safety architecture: three-layered permission system

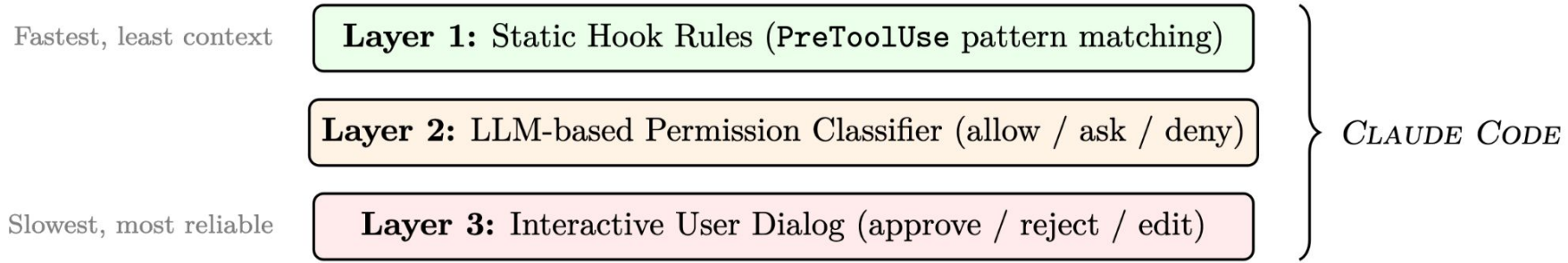


Figure 5: CLAUDE CODE's three-layered permission system.

Note: Sub-agents inherit the safety permissions of their parents, or are additionally restricted

(sub-agents cannot be given more permissions than their parent agent)

Safety in Codex

Utilizes a four-layer **permission stack**

(1) Execution policy

Static rules that decide whether execution needs review (no LLM involved)

(2) Lifecycle hooks

Project-level scripts that run static checks before execution

(3) Guardian

Separate approval LLM "Guardian" that evaluates tool/action risk and escalates high-risk cases to human approval

(4) Native OS Sandboxing

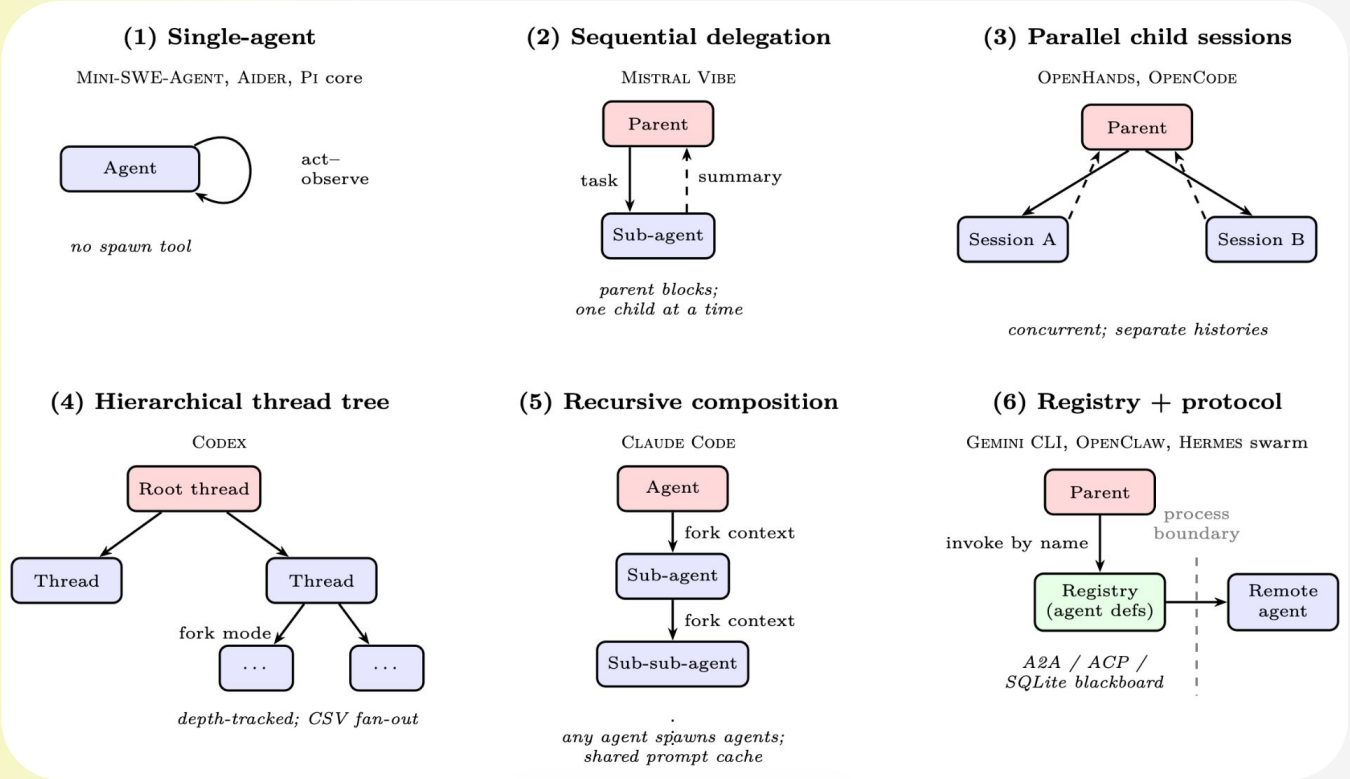
Uses built-in operating system or runtime features to isolate running code

Multi-agent orchestration (subsystem 6)

9 of 11 studied systems support multi-agent execution, but with varying architectures

This is the subsystem with the greatest diversity

Multi-agent taxonomy



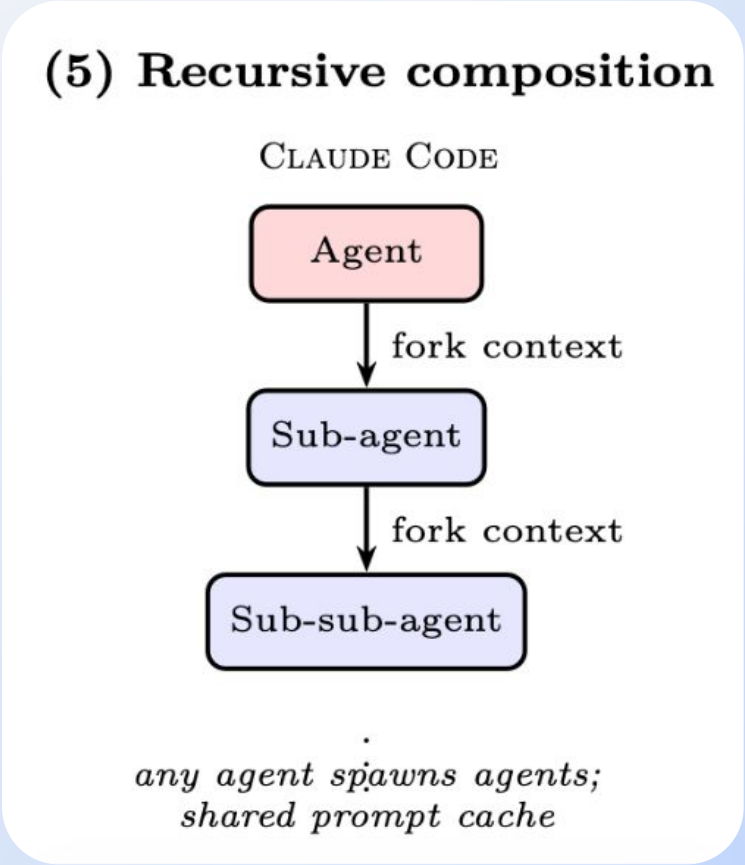
Recursive composition

Claude Code uses *recursive composition* for its multi-agent orchestration

This is the most sophisticated of the systems

Recursive composition is hierarchical

- The **overarching agent** breaks up the task into subtasks and spawns **subagents** to handle those subtasks
- Then each **subagent** can break the subtask further and spawn **subagents** as necessary for each task, etc
- This recursive system can lead to more clear, detailed specs for each task

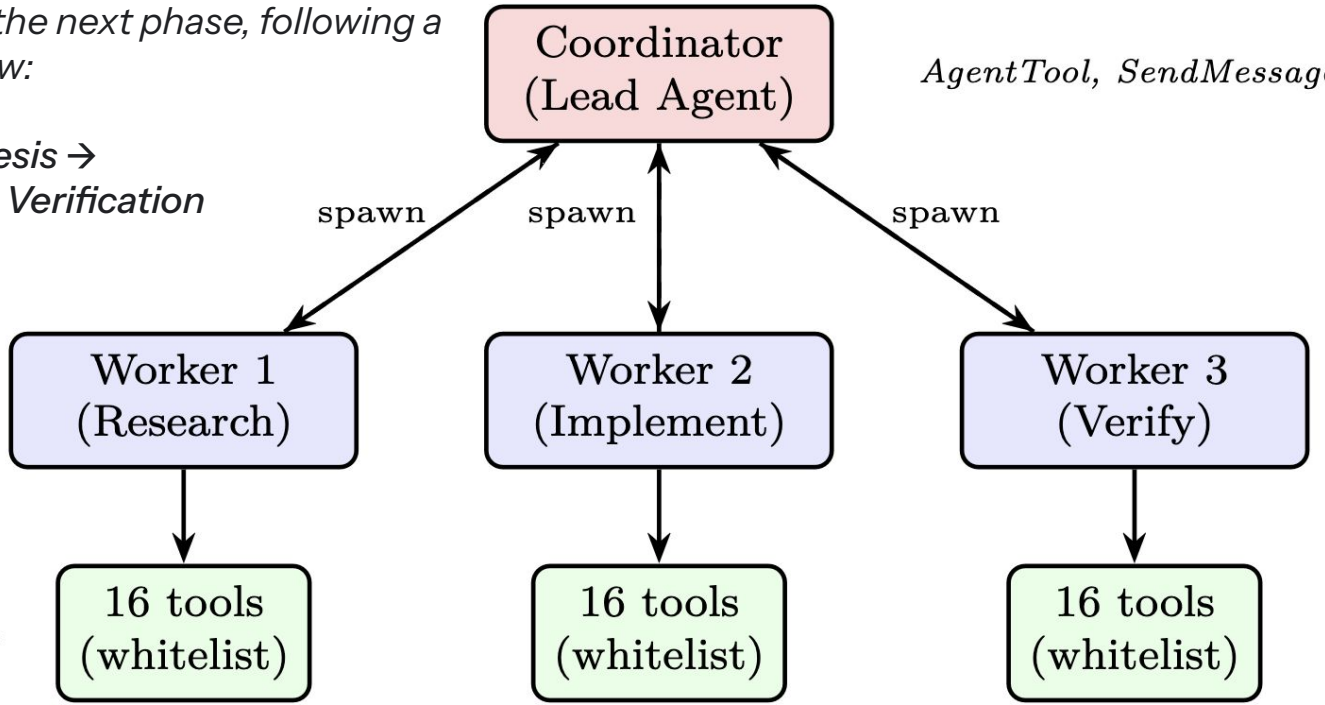


Coordinator mode

Also used by Claude Code—"a natural convergence for agents"

The coordinator synthesizes worker findings before delegating the next phase, following a structured workflow:

Research → Synthesis → Implementation → Verification



Extensibility (subsystem 7)

Each system in the corpus provides extension points, though through different mechanisms

Examples:

1. Model context protocol (MCP)

Extension mechanism that connects AI agents to external tools and workflows

MCPs standardize how agents interact with external tools: a *wire protocol*

8 of the 11 studied systems utilize MCP

2. Skills

Skills are "teachable units" in agentic flows (instructions, optional scripts, whitelists, metadata)

Skills standardize how agents package units of expertise into a discoverable directory: a *file-system convention*

9 of the 11 studied systems utilize skills

Conclusion

It's not all about the model—the **harness** is key to making a useful production system

Production-level systems invest in things that benchmarks do not always reward:

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

Takeaway: harnesses matter!

Thank you

What a Harness Is Not

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

A harness is not a **scaffold**—quite. The two words are near-synonyms in practice, and this paper inherits "scaffold" from its own earlier vocabulary. Where a distinction is useful, scaffold names the structural code (the loop, the registries) and harness names the shipped runtime artifact that embeds it: Mini-SWE-Agent's scaffold is 100 lines of Python; Claude Code's harness is a product with a terminal UI, a permission system, and a plugin ecosystem.

- A harness is not an **agentic framework**. A framework (LangChain, AutoGen, CrewAI) is a library the developer imports to build an agent; a harness is a runtime the developer works inside of. The distinction was clean in 2025 and is dissolving in 2026—from both directions—which is the subject of Section 14.2.

- A harness is not an **evaluation harness**. The SWE-bench "harness" wraps an agent to run it against tasks; an agent harness wraps a model to make it act. Same word, opposite direction of wrapping.

- A harness is not an **orchestrator**. An orchestrator (or meta-harness, Section 14.4) coordinates one or more harnesses from above and implements no editing loop of its own. Omnigent orchestrates eleven vendor harnesses—five of them systems in this paper—and implements no editing loop of its own; it is evidence about harnesses, not one of them.