The Harness Is Everything: What Cursor, Claude Code, and Perplexity Actually Built

Rohit
@rohit4verse
.
Mar 17
You are not using AI wrong because you haven't found the right model.
You are using AI wrong because you haven't built the right environment.
There is a reason some teams are shipping a million lines of code with three engineers while others are struggling to get a consistent refactor out of their agent pipeline. The difference is not GPT-5 versus Claude Opus. The difference is not the temperature setting or the max tokens. It isn't even the prompt, though everyone loses months of their life arguing about prompts.
The difference is the harness.
This article is about what that word actually means, technically and philosophically, because the industry has developed a bad habit of using it loosely. A harness is not a system prompt. It is not a wrapper around an API call. It is not an eval framework or a prompt template or a chatbot with memory. A harness is the complete designed environment inside which a language model operates, including the tools it can call, the format of information it receives, how its history is compressed and managed, the guardrails that catch its mistakes before they cascade, and the scaffolding that allows it to hand off work to its future self without losing coherence.
When you look at what Anthropic built to make Claude Code actually work, what OpenAI built to ship a million lines of code through Codex with zero manually-written code, and what the Princeton NLP group published in their landmark SWE-agent paper about agent-computer interfaces, you start to see the same pattern emerging from every serious team working in this space.
The model is almost irrelevant. The harness is everything.
This is a detailed technical breakdown of how that idea became the defining insight of applied AI engineering in 2025 and 2026. It covers the research, the real implementations, the failure modes

that motivated the design decisions, and the patterns that repeat whether you are building a coding agent, a research agent, or a long-running autonomous software engineer. By the end, you will understand not just what a harness is, but why building one correctly is now the most valuable engineering skill in the industry.

Part One: The Problem Nobody Talks About

Why Raw Capability Is Not Enough

In mid-2024, something strange happened in AI benchmarks. Researchers started noticing that the same frontier model could produce wildly different results on identical coding tasks depending entirely on how the task was presented and what tools were made available. The model had not changed. The underlying intelligence had not changed. What changed was the interface.

This should not have been surprising. We have known for decades that the right tools make engineers dramatically more productive. A software developer with a modern IDE, debugger, version control, and CI/CD pipeline is orders of magnitude more effective than the same developer working in a raw terminal with only a text editor. The IDE does not make the developer smarter. It removes friction, surfaces information at the right moment, catches errors early, and organizes work into navigable units.

Language models are the same. They are not general reasoners working from some infinite internal knowledge base. They are sophisticated pattern-matching engines that operate on tokens in a context window. Everything they know in a given moment is determined by what is in that context window, and everything they produce is conditioned on how that context is structured. The format of the input is not decoration. It is the cognitive architecture of the agent.

The interface is not a convenience layer. For an LM agent, the interface is the mind.

This is the central claim of the SWE-agent paper published by the Princeton NLP group in 2024, and it holds up under scrutiny. The paper introduced the concept of an Agent-Computer Interface (ACI) and demonstrated that a carefully designed ACI could produce a 64% relative improvement in benchmark performance compared to the same model interacting through a standard Linux shell. Same model, same task, same compute budget. The only variable was the interface.

Let that land for a moment. 64% is not a marginal gain. That is the difference between a tool that works and a tool that does not. And it came entirely from environment design, not from any improvement in the underlying model.

The Context Window Is Not a RAM Slot

The naive mental model of an AI agent treats the context window like RAM. You load data in, the model processes it, you get output. More context equals better performance. Longer prompts equal richer understanding. This mental model is wrong in ways that will ruin your agent if you build around it.

The context window is actually closer to the agent's entire working consciousness for a given session. Every token in that window costs computation. Every irrelevant piece of information competes for attention with the relevant information. The model does not have a selective attention

mechanism that cleanly ignores noise. The noise is in the room, and it affects the reasoning.
This has specific, measurable consequences for agent design. When you run grep on a large codebase from inside an agent loop and return ten thousand lines of matches, you have not given the agent more information to work with. You have flooded its working memory with irrelevant data that will degrade the quality of every subsequent step until the context is cleared. When you dump an entire file with cat because the agent wanted to see two functions, you have handed it a firehose when it needed a drinking glass.
The SWE-agent researchers were meticulous about documenting these failure modes. A standard bash interface caused agents to thrash. They would issue grep commands that returned thousands of lines, lose track of what they were looking for, issue more grep commands, gradually fill up their context with noise, and eventually either produce a wrong answer or stop making progress entirely. The problem was not model intelligence. The problem was that the interface had no mechanism for protecting the agent from itself.
The ACI solution was to build a search tool that returned a **capped, summarized list** of results. If your search returned more than 50 matches, the tool would suppress the output and tell the agent to narrow its query. This single design decision, which looks almost insultingly simple in retrospect, was one of the highest-leverage changes in the entire paper. It transformed a context-flooding failure mode into a natural refinement loop.

Part Two: The SWE-Agent Paper and the Birth of the ACI

What an Agent-Computer Interface Actually Is

The ACI is defined in the SWE-agent paper as an abstraction layer situated between a language model agent and a computer environment. The analogy to a human-computer interface (HCI) is intentional. Just as HCI research asks how to design interfaces that match human cognitive architecture, ACI research asks how to design interfaces that match LM cognitive architecture. Human cognitive architecture involves visual pattern recognition, spatial memory, parallel attention across a screen, and the ability to skim and selectively focus. LM cognitive architecture is fundamentally different. It involves sequential token processing, sensitivity to context order and formatting, limited working memory, and a tendency to anchor on whatever information appears most prominently in the prompt. Designing a good ACI means understanding these constraints and building around them, not against them.
The SWE-agent ACI for coding tasks had four main components, and each one reflects a specific insight about how language models fail when given raw computer access.

Search and Navigation

The search component replaced standard grep and find commands with purpose-built tools: find_file, search_file, and search_dir. The key difference was not syntax. The key difference was output management. Results were capped at 50. If a query exceeded that limit, the tool returned a message explaining that there were too many results and prompting the agent to refine its search. This sounds trivial. In practice, it was one of the most consequential decisions in the paper.

The reason it matters is that agents, like humans under cognitive load, tend to keep doing what they are doing when they feel uncertain. When a human is lost in a large codebase, they search more and more broadly, generating more and more noise. The capped search tool interrupted this pattern by creating a forcing function. You cannot proceed by being vague. You must be specific. This pushed the agent toward more deliberate, targeted behavior.

The File Viewer

The file viewer is where the paper's insights about cognitive architecture get most concrete. The researchers tested multiple viewer configurations and found that showing 100 lines at a time was a Goldilocks number. Fewer lines (they tested 30) caused agents to lose context about the surrounding code and make editing mistakes. More lines (or the full file) caused agents to lose track of where they were and miss important details.

The viewer was stateful. It maintained a position in the file across interactions. And critically, it prepended explicit line numbers to every visible line. This last detail sounds cosmetic. It was not. When an agent needs to issue an edit command targeting lines 47 through 52, it needs to be able to read those numbers directly from the view rather than counting them or performing arithmetic. Removing that cognitive task from the agent's working memory freed up capacity for the actual problem-solving.

The File Editor With Linting

The file editor's key innovation was immediate feedback with guardrails. The edit command accepted a start line, end line, and replacement text as a single operation. After every edit, the tool automatically ran a linter on the modified file and reported the result. If the edit introduced a syntax error, the edit was rejected before it was applied, and the agent received a clear error message showing both the original code and the failed edit.

This closed the feedback loop that causes cascading failures in naive agent implementations. Without a linter, an agent can introduce a syntax error, run the test suite, see a failure that seems unrelated (because the real error is elsewhere), spend multiple steps chasing the wrong problem, and eventually exhaust its context window chasing a ghost. With the linter integrated directly into the editor, syntax errors are caught at the moment of introduction, and the fix is localized before the problem can propagate.

Compare this to what happens when an agent uses raw bash tools. With sed or output redirection, there is no integrated feedback. Edits execute silently. Multi-line changes require complex argument formatting that is highly prone to mistakes. The agent might successfully run the command and introduce a subtle formatting error that the linter would have caught, and then spend the next ten steps wondering why the tests are failing.

Context Management

The fourth component addressed a problem that compounds over long sessions: the accumulation of stale context. As an agent works through a task, its history fills up with old observations, intermediate states, and exploratory steps that no longer reflect the current state of the

environment. All of that history takes up space in the context window and can actively mislead the agent by providing outdated information.

The ACI's context management system collapsed older observations, those beyond the last five turns, into single-line summaries. This kept the active context focused on recent, relevant information while preserving a compressed record of the overall trajectory. The agent could always see what it had recently done and what the current state was, without being buried in the full uncompressed history of every command it had ever run.

The Benchmark Results and What They Actually Mean

The SWE-agent paper benchmarked its ACI against the SWE-bench dataset, a collection of real GitHub issues from popular Python repositories. The task is to take a natural language bug report and produce a code patch that resolves it. This is a hard, practical task that requires navigating unfamiliar codebases, understanding error messages, writing correct code, and verifying the fix.

Using GPT-4 with a standard bash shell interface, the system resolved 3.97% of issues. Using GPT-4 with the purpose-built ACI, the system resolved 12.47% of issues. That is the 64% relative improvement mentioned earlier, and it came entirely from interface design.

The researchers also ran ablation studies, removing one component at a time to isolate the contribution of each design decision. The linter integration was consistently among the highest-leverage components. The capped search was critical for preventing context flooding. The stateful file viewer with line numbers meaningfully outperformed both the raw cat command and simpler viewer designs.

The performance difference was not about model intelligence. It was about cognitive load management. The ACI reduced the work the model had to do to track state, making room for the work that actually mattered.

The implications extend well beyond coding agents. Any long-horizon agent task involves the same fundamental challenges: navigating large information spaces, maintaining coherent state across many steps, catching and recovering from errors, and managing the limited resource of context window attention. The ACI design principles generalize. The specific tools change. The underlying architecture of the problem does not.

Part Three: Anthropic's Harness Engineering (The Long-Running Agent Problem)

Why the Context Window Boundary Is the Hard Problem

The SWE-agent paper addressed how to design the interface for a single agent session. Anthropic's engineering team, working on the Claude Agent SDK and Claude Code, encountered a different problem: what happens when a task is too large to complete in a single context window?

This is not a niche edge case. Most real software projects are too large to fit in any context window. A production web application has hundreds of files, thousands of functions, a test suite, configuration, documentation, and dependencies. Even with a 200K token context window, you cannot hold the full project in mind simultaneously. Human engineers solve this through external memory, documentation, version control, and the accumulated understanding that builds over

weeks and months of working in a codebase. An agent starting a fresh session has none of that.
The naive solution is compaction, and it works to a point. The Claude Agent SDK includes compaction capabilities that summarize old context when the window fills up. But compaction is not enough on its own. Anthropic's internal experiments showed that even with compaction, a frontier coding model like Opus 4.5 running in a loop across multiple context windows would consistently fail to build a production-quality web app from a high-level prompt.
The failures clustered around two patterns, and both of them are instructive.
The first failure pattern was attempting to do too much at once. When given a prompt like "build a clone of claude.ai," the agent would try to one-shot the entire application. It would begin implementing feature after feature without completing or testing any of them, run out of context window in the middle of implementation, and leave the next session to start with a half-implemented application, no documentation of what had been done, and no clear indication of what state the code was in. The next agent instance would spend most of its context budget trying to understand the mess rather than making progress.
The second failure pattern appeared later in projects. After some features had been built, a subsequent agent instance would look around, see that progress had been made, and conclude that the job was done. It would declare victory on a partially-completed application and stop working. This is not stupidity. It is a reasonable inference from incomplete information. The agent had no structured way to know what "done" actually meant for this project.
Both failures share a root cause: the agent had no persistent, structured understanding of the project's state that could survive the context window boundary and orient future sessions.

The Two-Agent Architecture: Initializer and Coding Agent

Anthropic's solution was a two-part architecture that has since become a template for how serious teams approach long-running agentic work.
The first part is an initializer agent. This is a specialized first session with a distinct system prompt whose entire purpose is to set up the environment that all future coding agents will operate in. It does not write features. It creates the scaffolding that makes feature development possible across many subsequent sessions.
The initializer agent produces three key outputs. First, it creates an init.sh script that can reliably start the development environment. This sounds mundane, but it has significant leverage. Every coding agent session that follows can begin by running init.sh rather than spending tokens figuring out how to start the servers, set up the database, and get the application into a testable state. Saving that overhead in every session accumulates.
Second, the initializer creates a comprehensive feature list file. In the claude.ai clone experiment Anthropic ran internally, this meant over 200 specific, end-to-end feature descriptions, things like "a user can open a new chat, type in a query, press enter, and see an AI response." Every feature was initially marked as failing. This file serves as the project's ground truth. A coding agent starting a new session reads this file and immediately knows, with certainty, what has been built and what

has not. It cannot look around, see some code, and conclude the job is done. The feature list tells it the truth.

Third, the initializer creates a claude-progress.txt file and makes an initial git commit. The progress file is a human-readable log that agents update at the end of every session, documenting what they worked on, what they completed, and what state they left things in. Combined with git history, this gives every future coding agent a fast way to orient itself without burning through its context budget on archaeology.

The second part is the coding agent. Every session after initialization uses a different prompt: work on one feature at a time, leave the environment in a clean state, and update the progress file and git history before the session ends. Incremental progress, documented state, clean handoffs.

The Feature List as a Cognitive Anchor

The feature list deserves special attention because it solves a problem that is easy to underestimate. Without it, an agent operating in a complex codebase must infer project completeness from the code itself. This inference is unreliable. Code can exist that is not functional. Functionality can exist that is incomplete. An agent that reads the code and reasons about what is done will get the wrong answer often enough to be a serious problem.

The feature list makes completeness explicit and unambiguous. Each feature has a passes field that is either true or false. An agent either updates this field after verifying a feature works end-to-end, or it does not. There is no ambiguity. There is no inference required. The ground truth lives in the file.

Anthropic made a deliberate decision to store this list as JSON rather than Markdown. The reason is behavioral. Empirically, models are less likely to inappropriately modify or overwrite JSON files compared to Markdown files. JSON has a rigid structure that resists casual editing. This is a small detail with real consequences: you want the feature list to be something agents update carefully, not something they casually rewrite when they feel like it.

{ "category": "functional", "description": "New chat button creates a fresh conversation", "steps": ["Navigate to main interface", "Click the 'New Chat' button", "Verify a new conversation is created", "Check that chat area shows welcome state", "Verify conversation appears in sidebar"], "passes": false }

The instruction accompanying this format was explicit: it is unacceptable to remove or edit tests because this could lead to missing or buggy functionality. You prompt the model to treat this file as inviolable. The JSON structure reinforces that instruction architecturally.

Incremental Progress and the Clean State Requirement

One of the hardest problems in multi-session agentic work is ensuring that each session ends in a state that the next session can safely build on. Without explicit enforcement, agents tend to leave work in whatever state they happen to be in when the context window fills up. Half-implemented features, broken tests, undocumented changes. The next agent inherits the mess.

Anthropic's solution was to make clean state a first-class requirement rather than a nice-to-have.

Every coding agent session ended with a git commit (with a descriptive message), an update to the progress file, and a reversion to a working state if needed. By "clean state" they meant code that would be appropriate for merging to a main branch: no major bugs, well-documented, in a state where a developer could reasonably begin a new feature without first untangling someone else's half-finished work.

The git commit was not just a checkpoint. It was a recovery mechanism. When an agent made a change that broke something, it could use git to revert to the last known-good state and try again. This is how human engineers work, and it turns out to be exactly the right discipline for agents too. Version control is cognitive scaffolding, not just source management.

Testing: The Failure Mode Nobody Likes to Talk About

Anthropic documented a failure mode that shows up in virtually every serious agentic coding project: agents marking features as complete without properly verifying them end-to-end. An agent would make a code change, run a unit test or a curl command against the development server, see a passing result, and mark the feature as done. But the feature would not actually work when tested through the browser as a user would.

The gap between unit test success and end-to-end functionality is something human engineers navigate by shifting contexts, by running the application and trying to use it. An agent without explicit browser testing capabilities has no way to perform this shift. It can only observe what its tools allow it to observe, and if those tools do not include browser automation, it will consistently miss a category of bugs that only manifest in real user flows.

The solution was to give agents access to the Puppeteer MCP server, a browser automation tool that allowed Claude to actually navigate the application, click buttons, fill forms, and verify that features worked end-to-end. The performance improvement was dramatic. Bugs that were invisible from the code alone became obvious when the agent could see what a user would see.

This is a concrete illustration of a general principle: the quality of an agent's work is bounded by the quality of its feedback loops. If your agent cannot observe the consequences of its actions in the domain that matters, it will optimize for proxy metrics that may not correlate with actual correctness.

The Startup Sequence: Getting Up to Speed Fast

Every coding agent session in Anthropic's harness began with a standardized startup sequence designed to orient the agent as quickly as possible without burning tokens unnecessarily. The sequence was:

Run pwd to confirm the working directory. Read the progress file and git log to understand recent work. Read the feature list and choose the highest-priority incomplete feature. Run the init.sh script to start the development environment. Run the basic end-to-end test to verify the application was in a working state.

Only after completing all of these steps would the agent begin working on a new feature. If the startup test revealed that the application was broken, the agent would fix the existing breakage

before touching anything new. This prevented the compounding problem where an agent starts a new feature on top of a broken foundation, making the underlying problem harder to isolate and fix.
The startup sequence also saved tokens in a specific way. Because the init.sh script documented exactly how to start the development environment, the agent did not need to figure it out from scratch. The tokens saved on environment setup in every session accumulate significantly over a long project.
[Assistant] I'll start by getting my bearings and understanding the current state of the project. [Tool Use] <bash - pwd> [Tool Use] <read - claude-progress.txt> [Tool Use] <read - feature_list.json> [Assistant] Let me check the git log to see recent work. [Tool Use] <bash - git log --oneline -20> [Assistant] Now let me check if there's an init.sh script to restart the servers. <Starts the development server> [Assistant] Excellent! Now let me navigate to the application and verify that some fundamental features are still working. <Tests basic functionality>

Part Four: OpenAI's Harness Engineering (Zero Lines of Manual Code)

The Experiment

In late August 2025, OpenAI's Codex team started a git repository with a single constraint: no human-written code. Every line of code in the repository, including application logic, tests, CI configuration, documentation, observability tooling, and internal developer utilities, would be written by Codex agents. Humans would steer. Agents would execute.
Five months later, the repository contained approximately one million lines of code across all of those categories. Roughly 1,500 pull requests had been opened and merged. A small team of three engineers had driven most of this, averaging 3.5 pull requests per engineer per day. As the team grew to seven engineers, the per-engineer throughput actually increased. The product had hundreds of daily internal users and external alpha testers.
This is not a demo. It is a real internal product built and shipped entirely through agent-generated code. The team wrote the article describing this experience in February 2026, and the central message is the same as the SWE-agent paper: the bottleneck was never model capability. The bottleneck was always environment design.

The Redefining of Engineering Work

The most important observation in OpenAI's harness engineering article is about how the engineering job itself changed. When your primary job is no longer to write code, what are you doing instead?
You are designing environments. You are specifying intent. You are building feedback loops. You are asking, constantly, not "how do I fix this bug?" but "what capability is missing from the environment that is causing this bug to appear?"
When something failed, the fix was almost never "try harder." It was almost always "what structural piece of the environment is missing or misconfigured that is causing the agent to fail here?" This is a profound shift in engineering thinking. You stop debugging code. You start debugging the system that produces code.

The primary job of the engineering team became enabling the agents to do useful work, not doing the work themselves.

In practice, this meant decomposing large goals into smaller building blocks, building the tools and abstractions that make those building blocks achievable, and using failures as signals about what the environment needed to better support. The human engineers worked depth-first: when an agent got stuck, they did not try to write the code themselves. They asked what was missing, built it into the environment, and let the agent try again.

Repository Knowledge as the System of Record

One of the most important architectural decisions in OpenAI's harness was making the repository itself the source of truth for everything an agent needed to know. The insight was simple but far-reaching: from an agent's perspective, anything it cannot access in context while running effectively does not exist. Knowledge that lives in Google Docs, Slack threads, or people's heads is invisible to the system.

Early in the project, the team tried the "one big AGENTS.md" approach. A single large instruction file containing everything the agent needed to know about the project, the architecture, the conventions, the constraints. It failed predictably, in four ways that are worth understanding.

First, context is a scarce resource. A giant instruction file crowds out the task, the code, and the relevant documentation. The agent either misses key constraints or starts optimizing for the wrong things. Second, too much guidance becomes non-guidance. When everything is marked as important, nothing is. The agent starts pattern-matching locally instead of navigating intentionally. Third, it rots instantly. A monolithic manual becomes a graveyard of stale rules as the codebase evolves. Fourth, it is hard to verify. A single blob does not lend itself to coverage checks, freshness tracking, or cross-linking. Drift is inevitable.

The solution was a structured docs/ directory treated as the system of record, with a short AGENTS.md file (roughly 100 lines) serving as a map that pointed to deeper sources of truth elsewhere. Design documentation was catalogued and indexed. Architecture documentation provided a top-level map of domains and package layering. Plans were treated as first-class artifacts with progress and decision logs checked into the repository.

This enabled what the team called progressive disclosure: agents started with a small, stable entry point and were taught where to look next, rather than being overwhelmed upfront. The result was that agents could reason about the full business domain directly from the repository, without needing access to external context that might not be available or might be out of date.

Application Legibility: Making the System Visible to the Agent

As code throughput increased, the bottleneck shifted from generation to verification. The team was generating code faster than human QA capacity could validate it. The solution was to make more of the verification work something agents could do themselves, by making the application directly legible to Codex.

This involved several concrete investments. They made the application bootable per git worktree, so

Codex could launch and drive an isolated instance of the application for each change it was working on. They wired the Chrome DevTools Protocol into the agent runtime and created tools for working with DOM snapshots, screenshots, and browser navigation. This enabled Codex to reproduce bugs, validate fixes, and reason about UI behavior directly, without requiring a human to interact with the application.

They built a full local observability stack: logs, metrics, and traces exposed to Codex via LogQL, PromQL, and TraceQL. Each agent task ran on a fully isolated version of the application with its own observability data, torn down once the task was complete. This meant agents could debug production-like issues using real observability tools, the same tools a human engineer would use, rather than having to infer behavior from the code alone.

The principle here is the same one the SWE-agent paper demonstrated: the quality of an agent's work is bounded by the quality of its feedback loops. If an agent can see what a user would see, and can observe the same metrics and logs a human engineer would observe, it can catch and fix a much broader class of problems than an agent operating on code alone.

Enforcing Architecture Without Micromanaging

One of the most interesting challenges in a fully agent-generated codebase is maintaining architectural coherence over time. Codex replicates patterns that already exist in the repository, including uneven or suboptimal ones. Over time, this leads to drift. Bad patterns spread. Inconsistencies accumulate. The codebase becomes harder for future agent runs to navigate correctly.

OpenAI's solution was to enforce invariants mechanically, not through human code review. The application was structured around a rigid architectural model: each business domain divided into a fixed set of layers with strictly validated dependency directions and a limited set of permissible edges. These constraints were enforced by custom linters (written by Codex, naturally) and structural tests.

The key insight was to enforce boundaries while allowing significant freedom within them. The linters checked that code flowed in the right direction through the layer hierarchy. They did not dictate how specific features were implemented within those boundaries. This is the same principle that makes platform teams effective at scale: enforce the foundation, allow autonomy on top of it.

The linters were custom-written specifically to generate helpful error messages for agents. When a linter caught a violation, the error message included remediation instructions formatted for injection into agent context. This closed the loop: the constraint violated, the rule that was violated, and the steps to fix it were all delivered in a single, actionable feedback message.

They also encoded what they called "golden principles" directly into the repository: opinionated, mechanical rules that kept the codebase legible and consistent for future agent runs. Prefer shared utility packages over hand-rolled helpers. Validate data shapes at the boundary. These principles were enforced by recurring cleanup background tasks that scanned for deviations, updated quality grades, and opened targeted refactoring pull requests. Most of these could be reviewed in under a

minute and automerged.

Throughput Changes the Merge Philosophy

When agent throughput dramatically exceeds human attention capacity, conventional engineering norms become counterproductive. Pull requests that sit waiting for review are blocking agent work. Test flakes that are investigated individually are consuming human attention that could be directed at higher-leverage tasks.

OpenAI's team made a deliberate decision to operate with minimal blocking merge gates. Pull requests were kept short-lived. Test flakes were addressed with follow-up runs rather than blocking progress indefinitely. When agent throughput far exceeds human attention, corrections are cheap and waiting is expensive. The right tradeoff looks irresponsible in a low-throughput environment and obvious in a high-throughput one.

This is a genuinely important insight for teams transitioning to agent-driven development. The merge philosophy that made sense when human engineers wrote every line of code does not automatically make sense when agents are generating 3.5 pull requests per engineer per day. The bottleneck shifts, and the process needs to shift with it.

Part Five: The Awesome Agent Harness Taxonomy

Mapping the Ecosystem

The Awesome Agent Harness repository, maintained by the AutoJunjie project on GitHub, attempts to map the emerging ecosystem of harness engineering tooling. Its central argument is worth stating explicitly before diving into the taxonomy: the ability of AI to write code is effectively a commodity. Foundation models can produce functional code. That is no longer the differentiating capability. The differentiating capability is coordination and environment design.

The repository catalogs the full stack of what a serious agent harness ecosystem requires, broken into seven distinct layers. Understanding these layers helps explain why "building an AI coding assistant" is actually many separate engineering problems, not one.

Layer 1: Human Oversight

At the top is the human oversight layer, where humans approve proposals, review pull requests, and set priorities. This is not a technical layer in the traditional sense. It is the interface between human judgment and agent execution. The key design principle here is that engineers should be designing environments and reviewing outcomes, not writing code directly. Their leverage comes from steering, not from executing.

Layer 2: Planning and Requirements (Spec Tools)

This layer translates human ideas into structured specifications and task DAGs (Directed Acyclic Graphs) that agents can consume reliably. The underlying insight is that agents execute blindly. If the specification is vague or ambiguous, the agent will produce something that satisfies its interpretation of the spec, which may not be what the human intended. Spec tools force precision at the requirements stage, before any code is written.

One project in this space, Chorus, attempts to solve what the repository calls the "reversed

conversation gap." Instead of having humans write detailed spec tickets (a major failure point because humans are not naturally precise in the way agents need them to be), Chorus lets the AI propose task DAGs and elaborate on requirements, with humans in a strict verification and approval role before execution begins. The AI is better at generating complete specifications from partial intent than humans are at writing them from scratch.

Layer 3: Full Lifecycle Platforms

These tools manage the end-to-end process from initial requirements to delivery, integrating AI proposals with human verification gates and sub-agent orchestration. They are the glue between the specification layer and the execution layer, handling state management across the full development lifecycle.

Layer 4: Task Runners

Task runners bridge the gap between issue trackers (GitHub Issues, Linear) and coding agents. The flow is: a human or PM agent creates an issue, the task runner spawns a workspace, the agent delivers a pull request, and the human reviews. Tools in this category include systems that continuously poll task queues, decide when to spawn agents, and deliver completed work without requiring human involvement in the execution loop.

Layer 5: Agent Orchestrators

Orchestrators solve the throughput problem by enabling parallel execution of multiple agents while isolating their work in separate git worktrees. This is critical because agents working in parallel on the same codebase will conflict with each other if they share a workspace. Git worktree isolation gives each agent its own sandbox, allowing many agents to work simultaneously without stepping on each other.

Tools like Vibe Kanban, Emdash, and Composio implement this pattern. Each agent task gets its own git worktree. Changes are validated in isolation before being merged. CI feedback, merge conflicts, and coordination between agents are all handled by the orchestration layer rather than requiring human intervention.

Layer 6: Agent Harness Frameworks and Runtimes

Frameworks provide composable primitives for building custom environments: progressive disclosure mechanisms, sub-agent spawning, structured context delivery. Runtimes provide persistent infrastructure: long-running memory, scheduled execution, multi-channel communication between sessions.

The distinction between a framework and a runtime is important. A framework is what you build on. A runtime is what keeps running. The Claude Agent SDK is primarily a framework. A system that runs agents on a cron schedule, maintains persistent memory across sessions, and handles multi-channel coordination between agent instances is a runtime. Both are necessary for serious long-running agentic work.

Layer 7: Coding Agents

At the bottom is the execution layer: Claude Code, Codex, and similar systems that write, test, and

debug code. The key insight of the repository is that this layer is a commodity. The agent's effectiveness is primarily determined by everything above it in the stack, not by the agent itself.
This is a provocative claim that the evidence supports. The SWE-agent paper demonstrated it empirically: same model, 64% performance improvement from interface design. OpenAI's Codex team demonstrated it operationally: the engineering work that mattered was environment design, not execution. Anthropic's harness engineering work demonstrated it practically: the initializer agent setup determined whether the coding agents could make progress at all.

Part Six: The Design Patterns That Repeat

Across all of these systems and all of these organizations, several design patterns appear repeatedly. They are not coincidences. They are engineering solutions to problems that emerge whenever you try to deploy agents reliably at scale.

Pattern 1: Progressive Disclosure

Do not give the agent everything it might need upfront. Give it the minimum it needs to orient itself and the pointers to find more when it needs it. This pattern appears in the SWE-agent's capped search (do not return all results, force the agent to refine), in OpenAI's docs/ architecture (a short map pointing to deeper truth), in Anthropic's startup sequence (read the progress file first, then the feature list), and in the harness frameworks that implement structured context layering.
The cognitive reason for this pattern is that context is a finite resource, and the agent's attention is not uniformly distributed across it. Information presented at the beginning of a prompt has disproportionate influence. A short, focused entry point that points to richer context elsewhere is more effective than a comprehensive dump that dilutes attention across everything.
The practical reason is maintenance. A short entry point that serves as a map to deeper documentation is something you can keep accurate. A monolithic document containing everything quickly becomes stale and counterproductive.

Pattern 2: Git Worktree Isolation

One agent, one worktree. This pattern appears in every serious orchestration system. The reasoning is straightforward: when you have multiple agents working in parallel (or when a single agent is running tasks in sequence), you need isolation between work streams. Without isolation, parallel agents will step on each other's changes. Even with sequential agents, you want the ability to validate changes in an isolated environment before they affect the main codebase.
Git worktrees provide this isolation at the filesystem level. Each agent gets its own working directory, its own branch, and its own environment. Changes are made in isolation, tested in isolation, and merged only when they pass validation. This is how modern CI/CD systems work for human engineers, and it turns out to be exactly the right model for agent orchestration as well.

Pattern 3: Spec First, Repository as System of Record

Agents are blind to informal knowledge. Anything that lives in a Slack thread, a Google Doc, or someone's head is invisible to the agent. The only thing the agent can work with is what is in its context window, and the only reliable source for that context is the repository.

This pattern shows up as the feature list file in Anthropic's harness, as the structured docs/ directory in OpenAI's system, as AGENTS.md files in various open-source frameworks, and as the spec tools layer in the awesome-agent-harness taxonomy. The common thread is that specifications, requirements, architectural decisions, and constraints must be encoded into machine-readable files in the repository before execution begins. If the agent cannot read it from the repo, it does not exist.

This has an important implication for how engineering teams should document their work. Documentation is no longer just for human readers. It is the mechanism through which human intent becomes legible to agents. Documentation that is ambiguous, stale, or stored outside the repository is documentation that actively impairs agent performance.

Pattern 4: Mechanical Architecture Enforcement

Human code review does not scale to agent-driven development. When an agent can open 3.5 pull requests per engineer per day, review cannot be the primary mechanism for maintaining code quality and architectural integrity. The solution is to encode architectural constraints as mechanical checks that run automatically.

Custom linters, structural tests, and CI pipelines replace much of what code review does in human-driven development. The advantage is that mechanical checks are consistent, fast, and provide immediate feedback at the point of violation. A linter that catches an architectural violation and returns a remediation instruction in the error message is more effective than a code reviewer who catches the same violation three days later in a pull request comment.

The key design principle is to enforce invariants, not implementations. You care deeply about dependency directions, boundary crossing, data validation at interfaces, and consistency in naming and structure. You do not care which specific library the agent uses or exactly how a function is decomposed, as long as it satisfies the behavioral contract. This gives agents significant autonomy within a well-defined structure.

Pattern 5: Integrated Feedback Loops

Every high-performing harness architecture closes the feedback loop as tightly as possible. Syntax errors caught by linters at edit time. Runtime errors surfaced through observability tools the agent can query. UI bugs caught through browser automation the agent can drive. Test failures returned with context about what broke and where.

The alternative, agents writing code that gets tested externally and produces failure messages that feed back in a later session, is slower, more expensive in tokens, and more likely to produce cascading failures. Every point in the feedback loop where the gap between action and consequence can be reduced is a point where agent performance can be improved.

This is the harness version of the classic software engineering principle about catching errors early. The earlier you catch an error, the cheaper it is to fix. For agents, this applies with even more force because errors that are not caught immediately accumulate in context and degrade the quality of subsequent reasoning.

Part Seven: What This Actually Means for Engineers

The Skill That Transfers

The harness engineering discipline is, at its core, systems thinking applied to agent environments. It requires you to understand the cognitive architecture of language models well enough to design environments that work with it rather than against it. It requires you to think about state management, feedback loops, error recovery, and context optimization in ways that are familiar from distributed systems engineering but applied to a new domain.

The engineers who are most effective in this emerging paradigm are not the ones with the best prompting skills, though prompting matters. They are the ones who understand how the whole system works: how context flows, where it gets corrupted, how feedback loops can be tightened, how state can be preserved across sessions, and how constraints can be enforced without micromanaging the agent's behavior.

These are not new skills in the abstract. They are extensions of skills that good software engineers already have. System design, API design, error handling, testing strategy. What is new is the domain: designing environments for LM agents rather than interfaces for humans.

The Questions You Should Be Asking

When you are building an agent system and something is not working, the harness engineering mindset produces a different set of questions than the naive mindset.

Instead of "how do I write a better prompt?" you ask "what information does the agent need that it currently cannot access?" Instead of "why is the model making this mistake?" you ask "what feedback loop is missing that would catch this mistake before it propagates?" Instead of "why is the agent not doing what I told it to?" you ask "what constraint in the environment is preventing the agent from doing what I told it to?"

This shift is not just semantic. It changes where you invest your engineering effort. Investing in a better prompt that solves this specific failure mode is local and temporary. Investing in a better tool that prevents a category of failure modes is general and permanent. The harness is where that permanent investment lives.

The Commoditization of Execution

There is an uncomfortable implication in the awesome-agent-harness repository's central argument that deserves to be stated plainly. If the execution layer is a commodity, then the long-term competitive moat in AI-driven development is not in the model. It is in the harness.

This means that organizations and individuals who invest in harness engineering, in building the scaffolding, the feedback loops, the observability, the spec tooling, and the orchestration that allows agents to do reliable work at scale, will have a durable advantage over those who are focused primarily on which model to use or how to prompt it.

OpenAI's Codex team built the equivalent of a custom development platform for their specific codebase and domain. Anthropic built a harness architecture that enables months of incremental progress on complex applications. The SWE-agent team built an interface that produces 64% better

results from the same model. None of these advantages came from the model. They all came from the environment.

The model is what thinks. The harness is what thinks about. Getting that distinction right is the entire game.

Part Eight: Building Your Own Harness

The Minimal Harness

You do not need to build OpenAI's observability stack or Anthropic's full two-agent architecture to benefit from harness thinking. The minimal effective harness for a coding agent on a real project has a small number of essential components.

Start with a persistent progress file. Something the agent reads at the beginning of every session to understand what was done last time, and writes at the end of every session to document what it did. This single change prevents the "declare victory too early" failure mode and ensures continuity across context window boundaries.

Add a structured task list. Not a vague description of the project, but a specific, enumerated list of verifiable completion criteria. Each item should describe a user-visible behavior that can be tested end-to-end. Mark each item with a status that the agent updates only after verification. This prevents the "partially done looks done" failure mode.

Add version control with descriptive commit messages as a first-class part of every session. Every session ends with a commit. The agent should not consider its work done until the code is committed and the progress file is updated. This creates the clean handoff that makes multi-session work coherent.

If you are building a web application, add browser automation. The difference between an agent that can only read code and an agent that can actually use the application it is building is the same as the difference between a developer who can only read code and a developer who can run the application. Most of the bugs that matter are only visible at runtime.

The Environment Audit

If you already have an agent system and it is underperforming, the harness engineering approach suggests a specific diagnostic process. Rather than reaching for a better model or a longer prompt, you do an environment audit.

Ask: what information does the agent need that it does not currently have access to? Where are the points in the task flow where the agent regularly gets stuck or makes mistakes? What feedback is missing that would allow the agent to catch those mistakes itself? Where is context getting polluted with irrelevant information? What constraints need to be enforced that are currently relying on agent judgment?

Each of these questions points to a specific harness improvement. Missing information becomes a new tool or a new document in the repository. Missing feedback becomes a new test, linter, or observability integration. Context pollution becomes a new context management strategy. Unenforced constraints become new mechanical checks.

This is the virtuous cycle of harness development: every failure is a signal about what the environment needs, and every improvement to the environment reduces the frequency of that failure across all future agent sessions.

The Last Thing

There is a pattern in how transformative technologies get misunderstood in their early phases. The thing that captures public attention, the raw capability, the impressive demo, the benchmark score, is rarely the thing that determines who wins in the long run. The infrastructure layer, the harness, the environment, is usually where the real value gets created and captured.

The web was transformative not because HTML existed but because search engines and browsers made the web navigable. Mobile was transformative not because smartphones existed but because app stores and developer tools made it possible to build on top of smartphones at scale. In both cases, the platform layer that organized the underlying capability was where the durable value lived.

AI agents are following the same pattern. The capability exists. The question is who builds the environments that make the capability reliable, controllable, and continuously improvable. The SWE-agent researchers understood this in 2024 and demonstrated it quantitatively. Anthropic understood it building Claude Code and documented it openly. OpenAI understood it building their internal product and shared the lessons. The awesome-agent-harness community is cataloging it across dozens of tools and frameworks.

The harness is everything. The model is the reasoning engine. The harness is the context, the constraints, the feedback loops, the memory, the tools, and the scaffolding that determines what the reasoning engine can actually accomplish. Getting the harness right is not a prompt engineering problem. It is a systems engineering problem. And it is the most important engineering problem in applied AI right now.