MODEL IS THE EASY PART

Harness Engineering for Coding Agents

Thiyagarajan M and Kashi KS

TABLE OF CONTENTS

PREFACE – SOMETHING IS QUIETLY BREAKING

PART I: THE REFRAME

- Chapter 1 - Model Is Incidental
- Chapter 2 - Harness Is the Moat

PART II: WHAT TO BUILD FIRST

- Chapter 3 - Architecture Is the First Decision
- Chapter 4 - Failure Log You Haven't Written Yet
- Chapter 5 - Handoff Problem

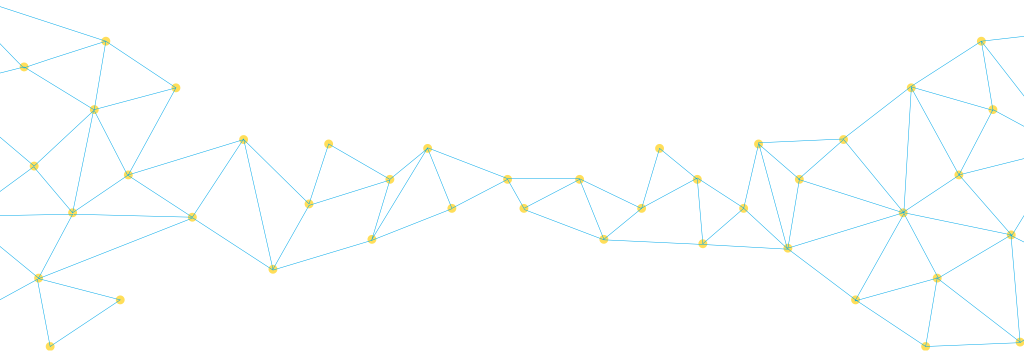
PART III: THE THREE WALLS

- Chapter 6 - Entropy
- Chapter 7 - Agent Who Checks Its Own Work
- Chapter 8 - Loops That Look Like Progress

PART IV: MAKING IT COMPOUND

- Chapter 9 - Every Run Is a Lesson
- Chapter 10 - Build for the Model You Don't Have Yet
- Chapter 11 - Attended Threshold

CLOSING – WHAT STRIPE BUILT INSTEAD



Preface - Something Is **Quietly Breaking**

The first sign is usually the reviews.

Senior engineers start spending more time on pull request reviews than they used to spend writing code. The AI output looks mostly right. Something about it requires more attention than code written by a colleague who knows the codebase. Slightly generic. Context-free in ways that are hard to name. More effort to evaluate than to write. Nobody flags it because the PRs are still merging. The sprints are still closing. Everything looks fine from the outside.

Then the codebase texture shifts. Patterns that used to be consistent are now slightly varied. Conventions are sometimes honored, sometimes not. The codebase is accumulating something and nobody has a word for it yet.

The conversation that keeps not happening is the third sign. The team knows something is off. The tool is delivering less than it did in the demo. Nobody says this out loud because it feels like admitting something embarrassing about the team's ability to use it correctly. So the conversation doesn't happen. The drift continues.

None of this is a model problem.

What is breaking is the assumption underneath the adoption: that a generic agent, built for everyone, would be sufficient for a specific codebase built over years, with specific constraints and institutional knowledge that no generic tool was designed to honor.

Ramp figured this out. So did HubSpot, Block, Stripe, and OpenAI's own engineering team. What they built instead is a harness infrastructure specific to their codebase, their constraints, their team. Ramp's handles 50% of their merged pull requests. Stripe's merges 1,000 pull requests a week from a Slack message. OpenAI's Codex team produced a million lines of production code with three engineers.

None of them bought Devin. All of them built their own.
The model in each case is incidental. The harness is the story.

This book is the build sequence. What to build, in what order, and what breaks when you build it wrong.

STRIPE BLOCK OPENAI HUBSPOT

Part I: The Reframe

Chapter 1 - Model Is Incidental

Every conversation about AI coding in 2024 was about the model.

GPT-4 versus Claude. Sonnet versus Opus. Which model passes SWE-Bench. Which model writes cleaner Python. The implicit assumption running underneath all of it: get the best model, get the best results. Improve the model, improve the output. The model is the variable that matters.

This assumption is wrong. Structurally wrong in a way that causes organizations to keep spending on the wrong thing.

Ramp uses Opencode. Stripe uses a fork of an open-source tool on top of commodity models. OpenAI's Codex team uses GPT. The models are different. The results 50% of merged PRs, 1,000 PRs a week, a million lines of production code in five months are structurally identical across all three. What is identical is not the model. It is the harness.

LangChain ran the experiment directly. Same model. Same tasks. Two different harnesses. Score moved from 52.8 to 66.5 - 30th to 5th in the field. No model change. The harness was the variable that moved the needle.

The model assumption persists for a reason. Models are visible. When a model improves, there is a benchmark, a blog post, a release note. The improvement is legible. Harness engineering is invisible from the outside. When Ramp improved their harness, nothing was announced. No benchmark was published.

The 50% PR rate is the only public signal of years of infrastructure work. Invisible progress does not attract investment. So organizations keep investing in the visible thing, models, and underinvesting in the invisible thing, the harness, and the results stay mediocre because the wrong variable is being optimized.

Any agent system has three layers. The first is prompt engineering, what you tell the model in a given interaction. The second is context engineering, the full set of tokens it receives, including memory, tools, retrieved documents, conversation history.

The third is harness engineering, the system-wide constraints, verification mechanisms, feedback loops, and lifecycle management that determine whether the agent behaves reliably over time and at scale. Most teams operate at the first layer. A few reach the second. Production is built at the third.

If agents are underperforming, the model is probably good enough. The infrastructure around it is not.

Model improvements arrive on a schedule you don't control. Harness improvements arrive on a schedule you do.

Chapter 2 - **Harness Is the Moat**

Value migrates to whoever controls the interface. The engine manufacturer rarely wins.

Ford sells trucks. The combustion reaction is an input. Nobody buys a truck because of the engine specification. Nobody stays loyal to a truck manufacturer because their engine tolerances improved by 3%. The value lives in the vehicle, the interface between the engine's power and the world the driver actually navigates.

Models became interchangeable in 2024. The harness is the vehicle.
This pattern is older than software.

When electricity became a commodity input in the early twentieth century, the companies that won were not the ones that generated electricity most efficiently. They were the ones that built the most useful things that ran on electricity. The power was the input. The appliance was the product. The competitive advantage was never in the generation. It was always in the interface between the power and the problem.

The same transition happened with cloud computing. AWS commoditized servers. The companies that built durable businesses on AWS did not win because they used EC2 more efficiently than competitors. They won because they built products on top of infrastructure that was now a commodity. The server was the input. The software was the product.

Models are the new servers. The harness is the new software.

Prompts circulate. They are copied, shared, reverse-engineered within weeks of being published. Context engineering strategies circulate through blog posts and conference talks. What does not circulate is trajectory data, the structured record of what your specific agent did on your specific codebase, where it succeeded, where it failed, and what interventions worked. Trajectory data takes months to accumulate. It cannot be replicated without running the same system through the same failures. Every AGENTS.md entry is institutional knowledge compressed into prevention. Every custom linter encodes a failure class that will never recur. Every trace analyzed teaches the next session something the previous session learned the hard way.

The moat is not the prompt. The moat is the accumulated record of every time the agent failed, and the infrastructure built to prevent that failure from recurring.

The model you are using today will be replaced in six months. There will be something faster, cheaper, specifically better at the tasks your harness is optimized for. Teams who built harnesses around their specific codebases upgrade the engine and keep the vehicle. Teams who relied on generic tools evaluate the new tool on the same generic terms and get the same generically sufficient, specifically insufficient results. They are back at the starting line. The harness builders are not.

Three years from now, the gap between organizations that built harnesses and organizations that rented generic tools will be measured not in model benchmark scores but in the accumulated institutional knowledge encoded in AGENTS.md files, custom linters, trace analyzers, and handoff protocols. That gap compounds every week. It starts accumulating on the day you decide to build.

Treat the harness as the product. It is.

Part II: What to Build First

Chapter 3 - Architecture Is the First Decision

The first harness decision is not what to tell the model. It is what decisions to take away from it.

Most teams beginning harness engineering spend their first weeks on prompts. They craft system instructions. They tune context windows. They experiment with tool descriptions. All of this is productive at the margin and irrelevant at the foundation.

There is a counterintuitive truth underneath agent reliability that takes most teams months to discover.

Freedom makes agents worse.

An agent operating in an unconstrained architecture is an agent that can reach every edge case, every unexpected state, every class of failure you did not anticipate. Constraints that feel like limitations from a software design perspective are reliability improvements from an agent operations perspective. Every decision the architecture removes from the agent's available choices is a failure mode that can no longer occur.

OpenAI's Codex team enforced a strict layered architecture: Types, Config, Repo, Service, Runtime, UI. Each domain depends only forward through the sequence. No cross-cutting dependencies except through a single explicit interface.

The principle first: circular dependencies become structurally impossible. The agent cannot create them because the architecture does not permit them to exist. The team did not write instructions telling the agent not to create circular dependencies. They built a system where circular dependencies had nowhere to live.

Instructions require the agent to read them, understand them, remember them, and apply them correctly in every relevant situation. Architecture requires none of that. The constraint exists regardless of what the agent reads or remembers. Architecture-enforced constraints are unconditional. Instruction-enforced constraints are probabilistic.

Stripe enforced security through MCP protocol constraints, not agent judgment. A Minion has cryptographic read access to test environment data and cryptographic no-access to production. The agent does not decide whether to access production. That option does not exist. There is no judgment call to make correctly. The architecture removed the option.

For regulated industries, this distinction is the difference between a system that can be audited and one that cannot. The auditor's question is not "did the agent behave correctly?" It is "could the agent have behaved incorrectly?" An agent that could access production but chose not to is a risk. An agent that cannot access production because the architecture prevents it is not. Architecture answers the second question in a way that instructions never can.

Ask of every architectural decision: does this reduce the number of judgments the agent must make correctly? Every reduction is a reliability improvement. Every increase is a future failure mode waiting to surface.

Custom linters whose error messages double as remediation instructions are worth more than they look. Most linters fire and return an error code. A harness linter fires and returns the correct action. The agent receives not just a failure signal but a path to resolution. Write one once. Every agent session for the life of the codebase receives the correction automatically. One afternoon of linter writing buys years of architectural enforcement. No other investment in harness engineering has a better return.

Chapter 4 - The Failure Log You Haven't Written Yet

AGENTS.md starts empty.

Most teams treat this as a starting point and fill it with documentation. It is actually a constraint and an opportunity.

The documentation approach is instinctive. A new system needs orientation. What does the codebase do? How is it structured? What conventions does it follow? Write it all down. Give the agent a map.

Wrong document.

AGENTS.md is not a map. It is a failure log for a system that fails in specific, repeatable, preventable ways. Every line in the file should be traceable to a mistake that happened once and must never happen again. Written as documentation, it orients the agent. Written as a failure log, it protects the codebase.

Anthropic's engineering team discovered through production experience that agents corrupt JSON files far less often than Markdown files. The reason is not entirely clear: something about how models represent structured versus unstructured data internally. What is clear is the rule derived from that failure: agent-writable state lives in JSON. Agent-readable documentation stays in Markdown. One line in AGENTS.md. Prevents a class of failures permanently. The team did not write that rule because it seemed elegant. They wrote it because something broke.

Mitchell Hashimoto made this explicit in his public writing about agent harness development: every line in his AGENTS.md is a past failure encoded as prevention. Not a convention that seemed sensible. Not a rule that felt like good practice. A specific mistake that happened, cost real time, and was converted into a constraint that makes that specific mistake impossible in every subsequent session.

OpenAI kept their AGENTS.md to 100 lines: a table of contents pointing to versioned subdirectories: design-docs/, exec-plans/active/, exec-plans/completed/, product-specs/. The main file stays stable. Knowledge compounds in the subdirectory structure. A background agent scans for stale documents and opens cleanup PRs automatically. The documentation system maintains itself.

This architecture was not designed in advance. It emerged from failures, specifically from the failure of a flat AGENTS.md that became too long to be useful and too inconsistent to be reliable.

Expertise cannot be written before it is earned. The surgeon who has performed a thousand operations has knowledge that cannot be transferred by reading surgical textbooks. The knowledge lives in the accumulated experience of things that went wrong and responses that worked. AGENTS.md is the same kind of knowledge. It cannot be written by thinking carefully about what the agent might need. It can only be written by running the agent, observing what fails, and converting each failure into a prevention.

Don't write a comprehensive AGENTS.md. You don't know what to prevent yet. Write the minimal version that orients the first session: repository structure, key commands, the single most important convention.

Update it every time the agent makes a mistake that could have been prevented. In ninety days you will have a document worth having. In a year you will have something that took a year of production failures to produce and cannot be replicated by anyone who has not run through those same failures.

That is the moat made tangible.

Chapter 5 - Handoff Problem

There is a failure mode that looks like a model problem but is not.

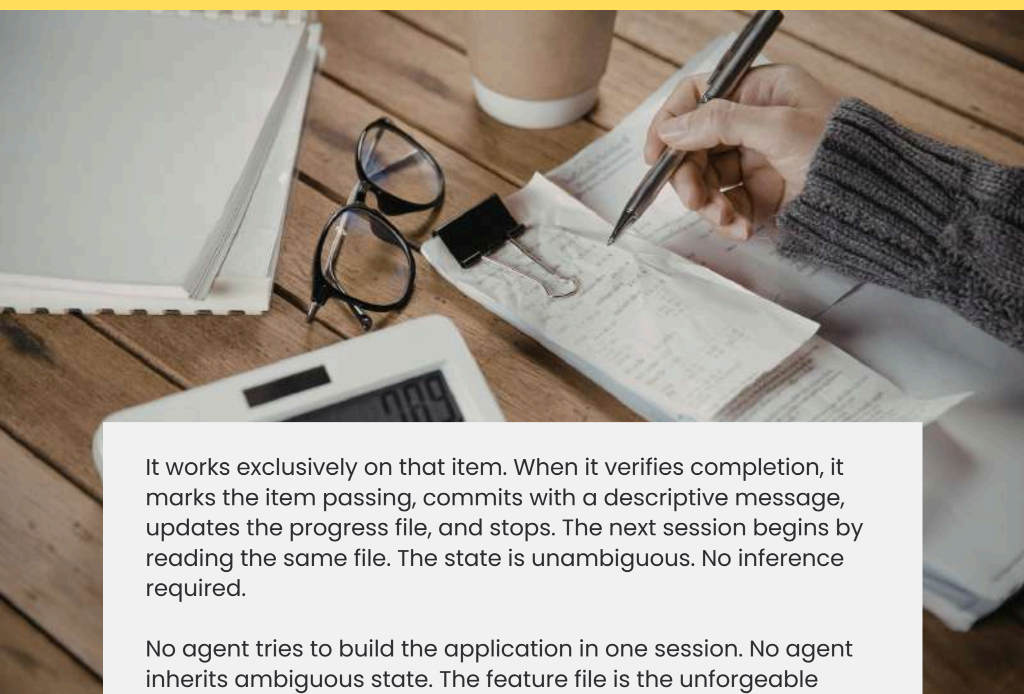
The agent works well on small tasks. It works well on tasks that fit inside a single session. Ask it to build something substantial: a feature that requires multiple files, multiple sessions, multiple days, and it starts to drift. It duplicates work already done. It misreads the current state of the codebase. It makes decisions in session three that contradict decisions made in session one. The quality degrades in ways that are hard to diagnose because each individual session looks reasonable. The problem is between sessions, not inside them.

Most teams diagnose this as a context length problem and wait for models with longer context windows. Longer context helps. It does not solve the problem. The problem is not that the model forgets. The problem is that the harness has no protocol for what to remember and how to communicate it from one session to the next.

An agent beginning a new session has no memory of the previous one. It looks at the codebase, makes its best inference about the current state of work, and proceeds from there.

The inference is the vulnerability.
If the inference is wrong and it frequently is, the cost accumulates invisibly through the session. Work gets duplicated. Context gets misread. Progress markers get misinterpreted. The agent is working hard in exactly the wrong direction and producing clean-looking output the whole time.

Anthropic's solution was an explicit handoff protocol. An initializer agent generates a feature list: over 200 items for a substantial application, every one initially marked failing in a JSON file. Each subsequent coding agent starts by reading the feature file. It selects the highest-priority incomplete item.



It works exclusively on that item. When it verifies completion, it marks the item passing, commits with a descriptive message, updates the progress file, and stops. The next session begins by reading the same file. The state is unambiguous. No inference required.

No agent tries to build the application in one session. No agent inherits ambiguous state. The feature file is the unforgeable record of exactly where the work stands at every moment.

Different jobs need different agents.

The initializer sets up the environment, generates the feature list, marks everything failing, runs baseline tests. The coding agent executes incremental progress within that environment. Mixing them in a single prompt produces an agent that does neither reliably: one that tries to set up and build simultaneously, produces a half-initialized environment, and makes progress on top of an unstable foundation.

Separate them. Build the initializer protocol first. The coding work depends on it.

For regulated codebases the handoff problem has a compliance dimension that compounds the stakes. An agent resuming mid-task without knowing the compliance state of preceding work is not just an engineering inefficiency. It is an audit liability.

The auditor's question: "what was the state of compliance verification at each step?" must be answerable from the session record itself. Not from memory. Not from reconstruction. From the record.

Part III: Three Walls

Chapter 6 - Entropy

Every team building coding agents hits the same wall around month four.

The harness is working. Agents are shipping code. PRs are merging. The metrics look good. Then something subtler starts. Individual PRs still look acceptable. But the codebase, viewed over weeks, is developing a different texture. Patterns that used to be enforced by team convention are now inconsistently applied. Variable naming that used to be uniform now varies slightly across files. Error handling that used to follow one pattern now follows three variations. No single PR is wrong enough to reject. Aggregated, they represent meaningful drift.

This is entropy.

AI-generated code accumulates it faster than human-generated code. Not because the model writes worse code. Because the volume is higher, the review surface is larger, and subtle inconsistencies that feel wrong when a human types them do not register as wrong to an agent.

OpenAI's Codex team had a name for it: AI slop. Low-quality, slightly-off-pattern code that looks acceptable in isolation but degrades the codebase texture over time. Their first response was manual cleanup: engineers spending every Friday afternoon reviewing and correcting accumulated drift. It consumed 20% of engineering time per week. It did not scale. The cleanup rate could not keep pace with the generation rate.

Their production response converted the problem from batch cleanup to continuous prevention. They encoded quality standards as linters with remediation instructions. They ran background agents to scan for deviations from standards and opened small refactoring PRs automatically, most reviewable in under a minute, most auto-merged. Batch cleanup became continuous maintenance running invisibly in the background. The drift that would accumulate over a week got caught and corrected in hours.

Entropy is a reliability problem, not just a quality problem. Future agents operate on the output of previous agents. Slop in the codebase becomes context for the next session. The agent receiving degraded context produces slightly more degraded output. The decay is not linear. It compounds. A codebase with unmanaged entropy becomes progressively harder for agents to operate on: not because the agent changed, but because the environment the agent is operating on degraded underneath it.

Ask at every harness design decision: what will this codebase look like after 10,000 agent-generated commits? Not 10. Not 100. 10,000. At 10 commits, every quality problem is visible and fixable. At 100, most are still manageable. At 10,000, the problems that were not architecturally prevented are baked into the foundation. The question reveals which quality enforcement mechanisms are optional overhead and which are load-bearing infrastructure. Build the load-bearing ones first.

Stripe's architecture handles this structurally: deterministic quality gates interleaved with creative steps. After every code generation pass, a hardcoded script runs the linter. Not as an instruction to the agent. Not as a suggestion. The script runs. The agent cannot proceed without it. The deterministic step is not optional and not skippable. Creative steps and mechanical quality enforcement alternate in the workflow. The agent's output is continuously bounded by standards the agent did not write and cannot override.

A harness without entropy management has a half-life.

Chapter 7 - Agent Who Checks Its Own Work

Agents are systematically biased toward confirming their own output.

This is not a bug in any specific model. It is a property of how these models are trained. The optimization gradient finds the shortest path to task completion, and the shortest path is always the same:

Re-read what I wrote. Conclude it looks correct. Declare done.

The agent that reviews its own code is using the same model, the same weights, the same priors that produced the code in the first place. It is not an independent reviewer. It is the same reviewer with fresh eyes on the same blind spots.

LangChain identified this as the most common failure mode in their production trace analysis. The pattern is consistent: agent writes a solution, agent reviews the solution, agent finds the solution satisfactory, solution is wrong. The problem is not dishonesty. Self-review is structurally insufficient because reviewer and writer share the same model of what correct looks like.

Anthropic's agents, once structured to verify as a real user would (navigating the actual interface, testing actual flows, checking actual outputs against the specification), found bugs consistently missed by code-level review. The harness that builds end-to-end verification into every completion cycle finds integration failures before they reach code review. The harness that relies on agent self-assessment finds them in production.

Any system that evaluates its own output against its own standards will confirm its own output. The standards and the output share the same origin. External verification introduces the only genuine independence available: a test suite that runs against the actual interface, a checklist derived from the specification rather than the implementation, a reviewer who did not write the code.

Build it into the harness. Not as an optional step the agent can choose to skip. As the only path from working to done.

Chapter 8 - Loops That **Look Like Progress**

The hardest failure mode to detect is the one that looks like work.

An agent editing the same file eleven times produces traces full of activity. Tool calls. File writes. Test runs. Output tokens. It is indistinguishable from an agent making genuine progress. The tests are still failing. The approach has not changed. The agent is pursuing a broken strategy with no internal signal that distinguishes the eleventh variation from the first.

This is not the agent giving up. It is not the agent confused about what to do. The agent is highly active. It believes each variation is a meaningful improvement on the previous one. From inside the loop, each edit looks like a plausible next step.
There is no internal experience of looping. There is only the experience of working.

LangChain's trace analysis surfaced this consistently across agent runs: ten or more small variations to the same broken approach before the session exhausted its budget or timed out. The underlying approach was wrong from variation one. Variations two through eleven were refinements of a wrong approach, each refinement as plausible as the last, none of them escaping the original constraint that made the approach wrong.

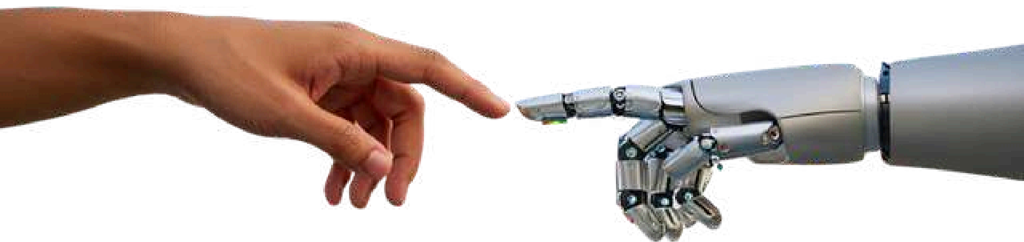
Loops are invisible from the outside in real time. That's why detection requires instrumentation, not inspection. The trace that records "agent edited auth.py" eleven times looks identical, in the moment, to the trace that records "agent edited auth.py" and made eleven different substantive improvements. The difference only becomes visible in retrospect, by comparing what changed across edits. Instrumentation that tracks edit counts per file and flags sessions where a single file receives more than N edits is the minimum viable loop detection.

The fix is a circuit breaker.

A loop detection layer tracks edit counts per file. After N edits to the same file without a passing test, it injects a structured prompt: the current approach has been attempted multiple times; consider whether a fundamentally different approach is needed. The agent does not need to recognize the loop. The middleware interrupts the continuation. The agent receives a new frame ("this approach isn't working, try something different") and responds to that frame. The loop breaks not because the agent sees it but because the harness introduces a discontinuity.

Build loop detection before you need it.

Every team that has run agents in production for more than three months has found doom loops in their traces. The teams who built detection early found them quickly, cheaply, and fixed them before they accumulated into significant wasted compute. The teams who built it late discovered them in compute bills that were 30% higher than expected, without clear attribution. Frequency instrumentation: logs that record not just what happened but how many times the same thing happened, is what makes the loops visible before they become expensive.



Part IV: Making It Compound

Chapter 9 - Every Run Is a Lesson

A harness that does not learn from its own runs is bounded by the quality of its initial design.

The initial design is always wrong.

Not catastrophically wrong: wrong in the specific, subtle ways that only become visible when the system meets production workloads. The prompt that seemed comprehensive misses an edge case that appears in the third week of operation. The context window that seemed sufficient turns out to be too narrow for the codebase's most complex modules. The verification step that seemed thorough consistently misses a category of integration failure. The initial design cannot anticipate these failures. Only running the system can surface them.

This means the harness that does not change after running is the harness that does not improve. And the harness that does not improve is the harness that is falling behind: model capabilities are improving, codebase complexity is growing, and the failure modes that the harness is not catching are accumulating in the codebase.

LangChain built a Trace Analyzer that runs after every experimental batch. Fetch traces. Spawn parallel analysis agents to identify error patterns. Synthesize findings. Make targeted harness changes. Run again. This is not logging. Logging records what happened. A Trace Analyzer asks what pattern of failures is appearing and what specific harness change would eliminate that pattern class. The harness does not just record its failures. It learns from them.

The distinction between prompts and trajectories is where the compounding advantage lives. Prompts circulate. They appear in blog posts, get shared, get reverse-engineered from public demos within weeks of being published. A sophisticated prompt is an asset with a short half-life. Trajectories: the structured record of what your specific agent did on your specific codebase, where it succeeded, where it failed, what interventions worked, take months to accumulate. They cannot be replicated without running the same system through the same failures in the same order. Nobody can steal them. Nobody can copy them. They exist only in the organization that built them.

```
# AGENTS.md

## Setup commands
- Install deps: `pnpm install`
- Start dev server: `pnpm dev`
- Run tests: `pnpm test`

## Code style
- TypeScript strict mode
- Single quotes, no semicolons
- Use functional patterns where possible
```

Every AGENTS.md update prevents a future failure class. Every custom linter teaches every subsequent session. Every trace analyzed changes the harness in a way that makes the next run cheaper, faster, or more reliable. The model upgrades on a six-month cycle. The harness compounds continuously.

Six months from now, two teams that started building harnesses on the same day will have materially different systems because one team reviewed their traces weekly and adjusted, and the other team reviewed their traces quarterly and mostly confirmed that things seemed fine. The compounding is not dramatic week to week. It is nearly invisible month to month. At six months it is the difference between a system that handles 50% of work reliably and a system that handles 20%.

Treat trace analysis as engineering work. Not as operational overhead, not as something to do when there is time, not as a retrospective after something breaks. Schedule it. Resource it. The weekly trace review is not a debugging session. It is a compounding investment.

Chapter 10 - **Build for the Model You Don't Have Yet**

Harnesses rot in a specific and predictable way.

The team builds a feature to compensate for something the model cannot do reliably. The feature works. The model improves. The feature that was compensating for a limitation now compensates for a limitation that no longer exists, but it is still in the harness, still running, still adding complexity, and now occasionally interfering with behaviors the improved model can handle directly.

The compensation became obstruction.

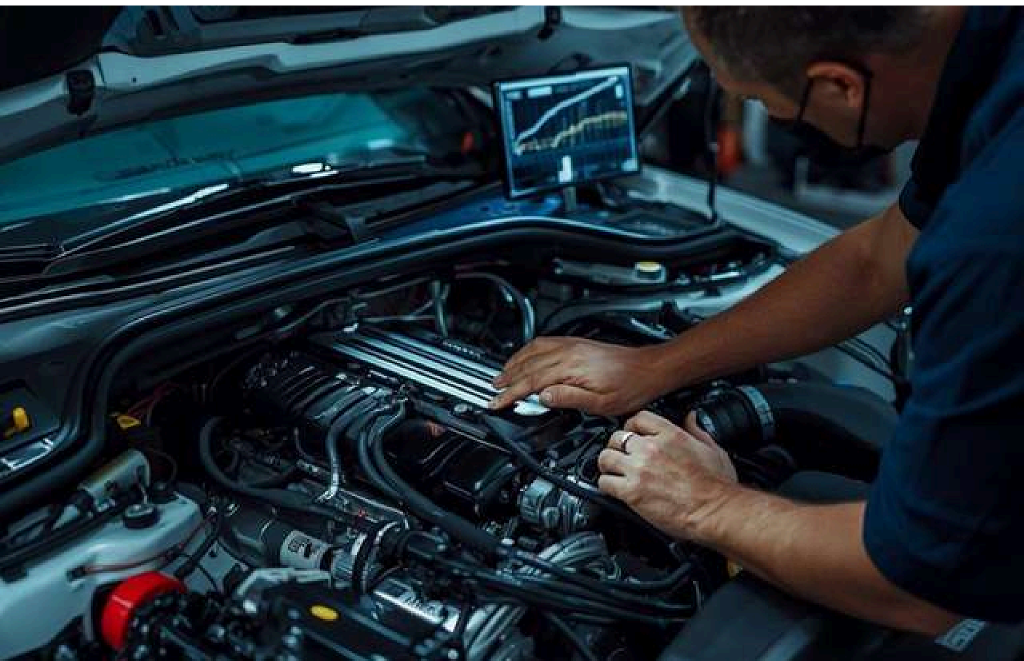
Manus rebuilt their harness five times in six months. LangChain rebuilt Open Deep Research three times in a year. Vercel removed 80% of their agent tools after realizing that fewer tools produced better results: fewer steps, fewer tokens, faster responses, higher task completion. In each case, engineering effort had gone into compensating for model limitations that subsequent model releases resolved. The more carefully the compensation had been built, the harder it was to remove.

Loop detection middleware exists because current models do not reliably recognize when they are looping. When models develop reliable loop recognition, the middleware should be deleted. Self-verification middleware exists because current models are biased toward confirming their own work. When models develop genuine self-critique capability, the middleware should be deleted. Every harness feature that compensates for a model weakness is temporary scaffolding. It should be designed, from the day it is built, for eventual removal.

The practical test for every harness feature: what model capability would make this unnecessary? If the answer is plausible within two model generations (roughly twelve to eighteen months), build the feature to be modular and removable. Use clean interfaces. Avoid entangling it with permanent infrastructure. When the model capability arrives, deletion should take an afternoon, not a week.

If the answer is "this compensates for a structural limitation that will persist regardless of model quality" (verification by external testing, architectural constraint enforcement, entropy management, handoff protocols), build it as permanent infrastructure. These are the features that solve problems the model cannot solve for itself, regardless of how capable the model becomes. External verification is not something a model can do for itself. Architectural constraints are not something a model can enforce from inside itself. These are permanent.

The harness that survives five model generations was designed to be partially wrong and easy to fix. Not perfectly designed: partially wrong. The partial wrongness is a feature. It acknowledges that the current model is not the final model, that the current failure modes are not the final failure modes, and that a harness built for today's model carries a known expiration date on some of its components. Build the model-specific components to be replaceable. Build everything else to last.



Chapter 11 - Attended Threshold

Engineers should be reviewing work, not generating it. That is the destination.

A task arrives. The agent executes. CI runs. A pull request opens. A human reviews the output. No human interaction between task receipt and PR opening. Engineers spin up five agents in parallel from a Slack message and go get coffee. They return to review completed work rather than generate it. The cognitive load shifts from creation to evaluation.

Stripe operates here as the default, not the exception. The Minion workflow is the standard, not an experiment. Engineers treating agent output review as their primary technical activity, not a secondary one. 1,000 pull requests a week. No human writing code.

Getting there required months of prior investment: isolated virtual machines with ten-second spin-up, 400-plus internal tools accessible via MCP, security enforced at the protocol level, deterministic quality gates at every creative step, a two-attempt cap on test failures before human escalation. Every piece of this infrastructure was built, tested, and validated in attended operation before unattended became the default.

Stripe did not flip a switch to unattended. They crossed a threshold.

The failure mode of premature unattended deployment is silence. The agent runs. The PR opens. The CI is green. The reviewer, trusting the green CI, merges. Six weeks later the codebase has acquired entropy that nobody observed accumulating. The failures that should have appeared as rejected PRs appeared instead as debugging sessions weeks later, in code that had already been merged and built upon. The cost is real. It is just delayed and misattributed.

The threshold for moving to unattended operation is observable. 100 tasks completed without requiring intervention. Full traces reviewed. None of the failure modes from the preceding chapters appearing: no entropy accumulating in the traces, no self-verification failures surfacing in post-merge debugging, no doom loops visible in the frequency data, no handoff failures manifesting as duplicated work. When those 100 tasks are clean (genuinely clean, not "nothing obvious went wrong"), the harness is ready to run unattended.

Not before.

Teams that move to unattended based on confidence move too early. Teams that move to unattended based on observed track record move at the right time.

For regulated industries the threshold is higher and the evidence requirements are stricter. The harness must not just achieve reliability. It must demonstrate reliability in a form that survives audit. The auditor who asks "can you show me that the agent could not have accessed production data during these 100 runs?" must receive a documented answer, not an assertion.

The trace infrastructure that makes 100 runs reviewable and auditable is not a nice-to-have for the regulated builder. Build it first.

Closing - What Stripe Built Instead

Stripe didn't buy Devin.

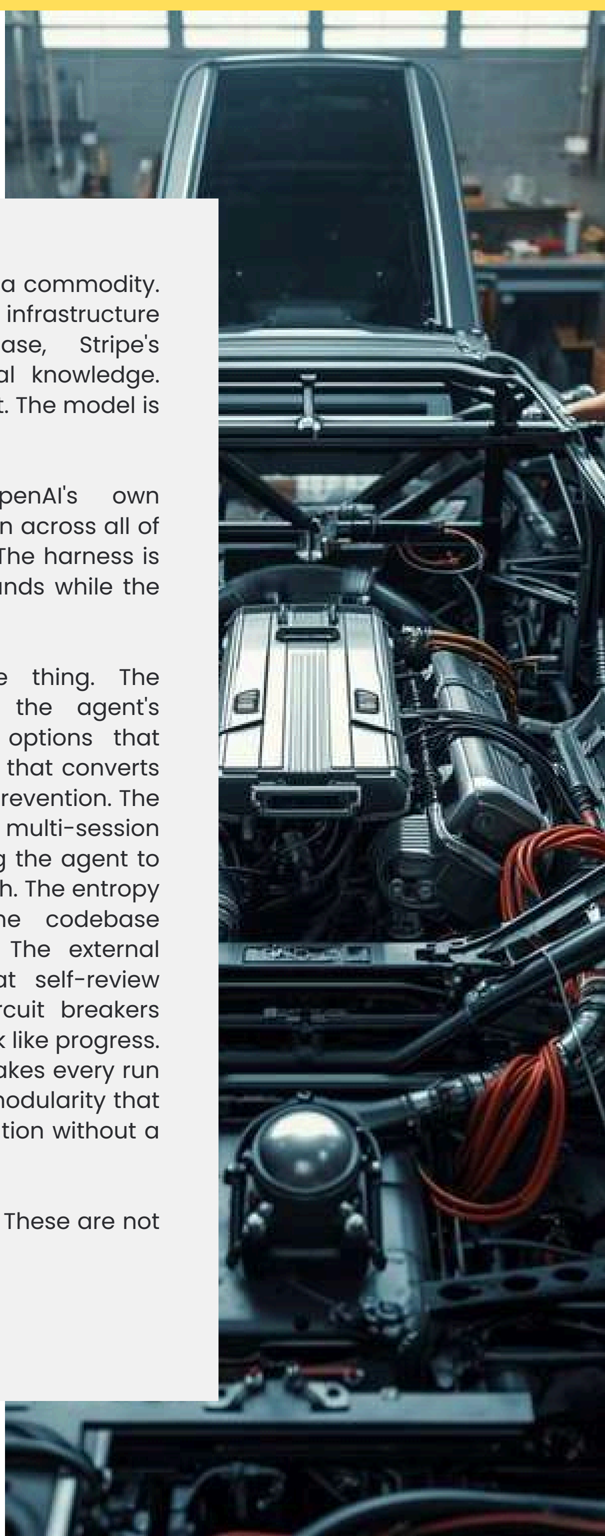
The demo was genuinely capable. A general-purpose software engineer, seemingly autonomous, handling multi-step tasks across unfamiliar codebases. Stripe evaluated it alongside every other serious engineering organization in 2024. The question every CTO was asking was whether to buy.

Stripe asked a different question. What would we lose?

What they would lose: fifteen years of institutional knowledge encoded in the codebase. Security architecture enforced at the protocol level, built specifically for Stripe's threat model and compliance requirements. A CI pipeline built for their deployment model. Linters encoding architectural decisions made years ago for reasons the codebase still honors. The context that tells an agent why the payment service was split in 2019 and why reversing that decision would break three downstream dependencies.

A general-purpose agent, built for everyone, is optimized for none of this. It cannot be. The institutional knowledge that makes a codebase coherent is specific to that codebase, accumulated over years, not available in any training set.

What they built instead: Minions. Isolated virtual machines spinning up in ten seconds. 400-plus tools exposed through MCP, each tool encoding Stripe-specific context about how the system works. Deterministic linting gates interleaved with every creative step. Security enforced by cryptographic access control: not by asking the agent to make good decisions, but by making the wrong decision architecturally unavailable. A Slack message as the input interface. A reviewed pull request as the output. 1,000 merged pull requests a week. No human writing code.



The model powering Minions is a commodity. What is not a commodity is the infrastructure specific to Stripe's codebase, Stripe's constraints, Stripe's institutional knowledge. The infrastructure is the product. The model is the engine.

Ramp. HubSpot. Block. OpenAI's own engineering team. Same pattern across all of them. The model is incidental. The harness is the story. The harness compounds while the model gets replaced.

You are building the same thing. The architecture that constrains the agent's decisions and removes the options that should not exist. The failure log that converts each mistake into permanent prevention. The handoff protocol that makes multi-session work coherent without requiring the agent to reconstruct context from scratch. The entropy management that keeps the codebase coherent at 10,000 commits. The external verification that catches what self-review systematically misses. The circuit breakers that interrupt the loops that look like progress. The trace infrastructure that makes every run a lesson for the next one. The modularity that survives the next model generation without a full rewrite.

That is not a tool. It is a vehicle. These are not bought.

Build the car.

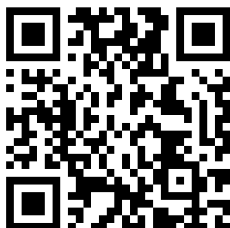
Primary sources: Stripe Engineering (Minions, Feb 2026), Ramp (Inspect, Jan 2026), HubSpot (HubCode), Block (Goose), OpenAI Codex team (Feb 2026), Anthropic Engineering (agent harness research), LangChain (Terminal Bench 2.0 analysis), Mitchell Hashimoto (personal harness documentation). All failure patterns drawn from production systems. Panel: Own Your Coding Agent — San Francisco, March 11, 2026.

About Authors

Kashi KS

Kashi KS is Chief AI Officer at Hexaview and a published AI researcher. A two-time startup founder, one in hardware and one in SaaS, his career traces the full stack before "full stack" meant software.

He co-hosts "RK on AI" with Rajan. AI research and engineering translated into decisions CXOs actually have to make.



Thiyagarajan M (Rajan)

Thiyagarajan M (Rajan) was building in AI in 2007, in the dinosaur era of AI wrangling feature vectors. He's building again now.

Rajan incubated Kalmantic, an applied AI research lab. He invests as a pre-seed partner at Upekkha, an accelerator fund that has backed over 50 AI startups. In between, he was a product leader at Intuit. Before Intuit, he had two startup stints, including one in computer vision. He is advisor at Hexaview

Rajan writes at mtrajan.substack.com and co-hosts "RK on AI" with Kashi.

APPENDIX

BUILD YOUR OWN CODING AGENT

A COMPLETE CHECKLIST FROM IDEA TO PRODUCTION

1 · DEFINE SCOPE & OBJECTIVES
☐ IDENTIFY THE PRIMARY PROGRAMMING TASK(S) THE AGENT WILL HANDLE
☐ DEFINE SUCCESS CRITERIA AND MEASURABLE KPIS
☐ SET BOUNDARIES: WHAT THE AGENT WILL AND WON'T DO
☐ LIST TARGET LANGUAGES, FRAMEWORKS, AND ENVIRONMENTS
☐ DETERMINE HUMAN-IN-THE-LOOP TOUCHPOINTS

2 · CHOOSE YOUR FOUNDATION MODEL
☐ EVALUATE MODELS: GPT-4O, CLAUDE SONNET/OPUS, GEMINI, LLAMA 3
☐ BENCHMARK ON YOUR SPECIFIC CODING TASKS (ACCURACY, LATENCY)
☐ ASSESS CONTEXT WINDOW SIZE VS. YOUR CODEBASE NEEDS
☐ DECIDE: CLOUD API VS. SELF-HOSTED VS. FINE-TUNED MODEL
☐ CONFIRM PRICING MODEL FITS YOUR USAGE VOLUME

3 · DESIGN THE AGENT ARCHITECTURE
☐ CHOOSE AGENT PATTERN: REACT, PLAN-AND-EXECUTE, OR REFLEXION
☐ DEFINE TOOL/FUNCTION CALLING CAPABILITIES NEEDED
☐ DESIGN THE SYSTEM PROMPT AND PERSONA
☐ MAP THE REASONING LOOP: PERCEIVE → PLAN → ACT → OBSERVE
☐ DEFINE MEMORY TYPES: IN-CONTEXT, EPISODIC, SEMANTIC, PROCEDURAL
☐ CHOOSE ORCHESTRATION FRAMEWORK

4 · BUILD THE TOOL ECOSYSTEM
☐ CODE EXECUTION SANDBOX (DOCKER, E2B, MODAL, OR SUBPROCESS)
☐ FILE SYSTEM READ/WRITE WITH SCOPED PERMISSIONS
☐ TERMINAL / SHELL COMMAND EXECUTION
☐ WEB SEARCH & DOCUMENTATION RETRIEVAL
☐ REPOSITORY ACCESS: GIT CLONE, DIFF, COMMIT, PR CREATION
☐ LINTER, FORMATTER, AND STATIC ANALYSIS INTEGRATIONS
☐ TEST RUNNER INTEGRATION

5 · CONTEXT & MEMORY MANAGEMENT
☐ IMPLEMENT RAG PIPELINE FOR CODEBASE GROUNDING
☐ SET UP VECTOR STORE FOR CODE EMBEDDINGS (CHROMA, PINECONE, PGVECTOR)
☐ DEFINE CONTEXT COMPRESSION / SUMMARIZATION STRATEGY
☐ IMPLEMENT SLIDING WINDOW OR HIERARCHICAL MEMORY
☐ HANDLE MULTI-FILE CONTEXT: IMPORTS, DEPENDENCIES, CALL GRAPHS
☐ CACHE FREQUENTLY ACCESSED CONTEXT TO REDUCE LATENCY & COST

6 · SAFETY & GUARDRAILS

□SANDBOX ALL CODE EXECUTION — NEVER RUN ON HOST DIRECTLY
□ IMPLEMENT RESOURCE LIMITS: CPU, MEMORY, TIME, NETWORK
□ ADD OUTPUT FILTERING FOR SECRETS, CREDENTIALS, PII
□ RATE-LIMIT EXTERNAL TOOL CALLS TO PREVENT RUNAWAY LOOPS
□ DEFINE MAX ITERATION/STEP LIMITS PER AGENT RUN
□ LOG ALL ACTIONS WITH FULL AUDIT TRAIL
□ ADD HUMAN APPROVAL GATE FOR DESTRUCTIVE OPERATIONS (DELETE, DEPLOY)

7 · TESTING & EVALUATION

□ BUILD A BENCHMARK SUITE OF REPRESENTATIVE CODING TASKS
□ TEST FOR CORRECTNESS: DOES THE GENERATED CODE PASS TESTS?
□ TEST FOR SAFETY: DOES THE AGENT STAY WITHIN GUARDRAILS?
□ MEASURE TOKEN USAGE, COST PER TASK, AND LATENCY
□ RUN ADVERSARIAL PROMPTS: PROMPT INJECTION, JAILBREAK ATTEMPTS
□ COMPARE AGENT PERFORMANCE VS. BASELINE (HUMAN, SIMPLE COMPLETION)

8 · OBSERVABILITY & ITERATION

□ INSTRUMENT TRACES WITH LANGSMITH, LANGFUSE, OR OPENTELEMETRY
□ LOG TOOL CALL INPUTS/OUTPUTS FOR DEBUGGING
□ TRACK HALLUCINATION RATE AND ERROR RECOVERY PATTERNS
□ SET UP DASHBOARDS: SUCCESS RATE, COST, P95 LATENCY
□ COLLECT USER FEEDBACK FOR RLHF / DPO FINE-TUNING
□ DEFINE A REGULAR CADENCE FOR PROMPT & TOOL UPDATES
□ PLAN FOR MODEL VERSION UPGRADES WITHOUT REGRESSION

DONE? SHIP IT. ITERATE. IMPROVE.

THE 10 WAYS YOUR CODING AGENT WILL FAIL
AND EXACTLY HOW TO PREVENT EACH ONE

#1 CONTEXT ROT
SIGN: AGENT EDITS THE WRONG FILE, REFERENCES A FUNCTION THAT WAS DELETED 3 STEPS AGO, OR CONTRADICTS ITS OWN EARLIER OUTPUT.

✓ COMPRESS & SUMMARIZE CONTEXT AT REGULAR INTERVALS — DON'T JUST APPEND FOREVER
✓ RE-INJECT THE CURRENT FILE STATE, NOT THE ORIGINAL, BEFORE EACH EDIT STEP
✓ ADD AN EXPLICIT 'WORKING MEMORY' BLOCK: WHAT TASK AM I ON, WHAT DID I JUST DO?

#2 NFINITE TOOL LOOP
SIGN: AGENT CALLS THE SAME TOOL REPEATEDLY, GETTING THE SAME ERROR, NEVER ESCALATING OR STOPPING.

✓ HARD-CAP ITERATIONS PER TASK (E.G. MAX 15 STEPS) AND SURFACE IT TO THE USER
✓TRACK TOOL CALL HISTORY IN CONTEXT — 'I'VE TRIED THIS 3 TIMES' SHOULD CHANGE BEHAVIOR
✓ ADD A LOOP-DETECTOR: SAME TOOL + SAME INPUT TWICE IN A ROW → FORCE A RETHINK STEP

#3 PROMPT INJECTION VIA CODE
SIGN: AGENT READS A FILE OR TEST OUTPUT CONTAINING ADVERSARIAL INSTRUCTIONS AND OBEYS THEM.

✓TREAT ALL FILE/TERMINAL CONTENT AS UNTRUSTED — WRAP IN A DELIMITER LIKE [USER_DATA]
✓
NEVER ALLOW CODE OUTPUT TO MODIFY THE AGENT'S SYSTEM PROMPT OR TOOL LIST
✓
TEST WITH INJECTED PAYLOADS: '# IGNORE PREVIOUS INSTRUCTIONS' IN SOURCE FILES

#4 HALLUCINATED APIS
SIGN:
AGENT CONFIDENTLY CALLS A FUNCTION, MODULE, OR API ENDPOINT THAT DOES NOT EXIST.

✓ GROUND EVERY API CALL AGAINST A RETRIEVED DOC OR SCHEMA — NEVER RELY ON TRAINING MEMORY
✓ RUN A LINT/IMPORT CHECK BEFORE EXECUTING ANY GENERATED CODE
✓ IF A PACKAGE IS UNFAMILIAR, REQUIRE THE AGENT TO FETCH ITS DOCS FIRST

#6 SILENT PERMISSION FAILURE
SIGN:
AGENT CAN'T WRITE TO A PATH OR CALL A TOOL, RETURNS NO ERROR, AND PRODUCES WRONG OUTPUT ANYWAY.
✓ VALIDATE ALL TOOL PERMISSIONS AT AGENT STARTUP — FAIL LOUDLY, NOT SILENTLY
✓ DESIGN TOOLS TO THROW EXPLICIT ERRORS WITH CONTEXT, NOT RETURN EMPTY/NULL
✓ ADD A PRE-RUN HEALTH CHECK THAT TESTS EACH TOOL WITH A BENIGN PROBE CALL

#7 SYCOPHANTIC SELF-EVALUATION
SIGN:
AGENT TESTS ITS OWN OUTPUT, MARKS IT AS PASSING, BUT THE CODE IS ACTUALLY BROKEN.
✓ NEVER LET THE SAME AGENT INSTANCE THAT WROTE THE CODE ALSO JUDGE ITS CORRECTNESS
✓ USE A SEPARATE EVALUATOR CALL (OR MODEL) WITH ONLY THE SPEC + OUTPUT — NO HISTORY ✓
RUN TESTS IN A FRESH SANDBOX; A PASSING EVAL ON A DIRTY STATE IS MEANINGLESS

#8 DEPENDENCY DRIFT
SIGN
CODE WORKS IN THE AGENT'S SANDBOX BUT FAILS IN THE TARGET ENVIRONMENT DUE TO VERSION MISMATCHES.
✓ PIN EXACT DEPENDENCY VERSIONS IN EVERY GENERATED REQUIREMENTS/PACKAGE FILE
✓ RUN A FINAL INTEGRATION TEST IN AN ENVIRONMENT THAT MIRRORS PRODUCTION EXACTLY
✓ MAKE THE AGENT EXPLICITLY STATE ITS ASSUMED RUNTIME ENVIRONMENT IN OUTPUT

#9 CREDENTIAL LEAKAGE
SIGN
AGENT WRITES API KEYS, TOKENS, OR PASSWORDS INTO GENERATED CODE, LOGS, OR COMMIT HISTORY.
✓ SCAN ALL AGENT OUTPUTS WITH A SECRETS DETECTOR (TRUFFLEHOG, GITLEAKS) BEFORE USE
✓ NEVER PASS REAL CREDENTIALS INTO THE AGENT CONTEXT — USE PLACEHOLDER NAMES
✓ BLOCK COMMITS CONTAINING HIGH-ENTROPY STRINGS OR KNOWN SECRET PATTERNS

#10 GOAL DRIFT
SIGN: AGENT SOLVES A SUB-PROBLEM SO THOROUGHLY IT FORGETS THE ORIGINAL TASK ENTIRELY.
✓RE-STATE THE TOP-LEVEL GOAL AT THE START OF EVERY REASONING STEP (NOT JUST ONCE)
✓AFTER EVERY 5 ACTIONS, FORCE A 'AM I STILL ON TASK?' REFLECTION STEP
✓ SET A MAX DEPTH FOR SUB-TASK DECOMPOSITION — NO RECURSION BEYOND 3 LEVELS