APRIL 2026 EDITION

Harness Engineering
Engineering Practices for the Age of AI Coding

From CLAUDE.md to a Complete Engineering System — A Practical Framework for Harnessing AI

The Complete Guide to AI Agent Harness Design

Version:v260411

Published:4/11/2026, 12:56:09 AM

Contents: Origins • Framework • 7 Case Studies • Hands-on • Reflections

Huashu

huasheng.ai

> From the "AI Coding Orange Book" Series

Table of Contents

CONTENTS

Part 1: Origins

§01 What Exactly Is a Harness?

§02 A Sixty-Year History: Humans and Their Tools

§03 Three Namings

Part 2: Framework

§04 Five Components of a Harness

§05 Less Is More: The Counterintuitive Art of Subtraction

Part 3: Case Studies

§06 OpenAI Codex Team: One Million Lines, Zero Handwritten

§07 Mitchell Hashimoto: One Mistake, One Rule

§08 Anthropic: Let AI Review AI

§09 Stripe Minions: The 1,300-PR-a-Week Pipeline

§10 LangChain: Same Model, Different Harness

§11 Kent Beck: An XP Pioneer's CLAUDE.md

§12 Huashu: From Zero Coding to a Million Users

Part 4: Hands-on

§13 From Scratch: Your First Harness

§14 The Instruction Layer: Give AI a Map, Not a Manual

§15 The Constraint Layer: Suggestions vs. Enforcement

§16 Capability and Memory

§17 Orchestration: Running Ten Horses at Once

Part 5: Reflections

§18 Experience Engineering: Who Will Design the Next Harness

A Note to Readers

Hi, I'm Huashu.

Let me start with the title. The word "harness" originally means horse tack, reins. A good horse needs a good set of reins before the rider can control it. AI is the same.

Over the past year and a half, I've been using AI coding tools almost every day. Building products, writing content, setting up workflows — AI is involved in everything. Along the way, I noticed a pattern: **most people use AI at the "chat" stage**. Think of something, ask a question, get an answer, ask again if it's not good enough. This works for simple problems, but it falls apart with complex projects.

What actually makes AI reliable isn't better prompts — it's a systematic collaboration framework. I call it a Harness: designing instructions for AI, setting boundaries, extending capabilities, managing memory, orchestrating workflows. Turning AI from a chat partner into a genuinely reliable engineering partner.

This isn't my invention alone. The book dissects 7 real case studies: how the OpenAI Codex team built a million-line product with zero handwritten code, how HashiCorp founder Mitchell Hashimoto trained AI with a "one mistake, one rule" approach, how Stripe's AI generates 1,300 PRs per week, and how extreme programming pioneer Kent Beck writes his CLAUDE.md. These people and teams take different approaches, but the underlying logic is the same: **don't expect AI to become smart on its own — you need to build it a harness**.

The second half of the book is a hands-on guide. Starting from a blank project, you'll build the instruction layer, constraint layer, capability layer, memory layer, and orchestration layer step by step, ultimately forming a complete Harness. You don't need to be an engineer — as long as you use AI tools, these methods apply.

Honestly, Harness Engineering as a concept is still in its early days. It doesn't have decades of experience and mature textbooks like software engineering does. But precisely because it's early, those who learn and practice now will have a first-mover advantage. As AI grows more powerful, the gap between those who know how to harness it and those who only chat with it will only widen.

This book is the third in the "AI Coding Orange Book" series. The series also includes: *Claude Code: From Beginner to Expert*, a practical starter guide; *Claude Code: Source Code Analysis*, which dissects Anthropic's architectural design; and *Agent Skills*, which teaches you to extend AI capabilities. The four books are independent, but if you want to get hands-on after reading this one, *From Beginner to Expert* is a great next step.

The core question this book answers is really just one: when AI can write code, what becomes the core human skill? My answer: the ability to design harnesses. I hope after reading this, you'll find your own

answer.

Huashu
April 2026

§01 What Exactly Is a Harness?

Harness Engineering: Definition and Origins

Yet another XX Engineering? Before you roll your eyes, let's look at where this word comes from and why it's precise.

Not Again?

Vibe Coding isn't even learned yet, Prompt Engineering courses are still being sold, and the ink on Context Engineering hasn't dried. Now here's another one.

Over the past year, four or five "XX Engineering" terms have popped up. The AI world coins new terms almost as fast as models iterate. Eye-rolling is a perfectly reasonable reaction.

But this time is different.

This isn't a term some KOL conjured up on Twitter in a flash of inspiration. **OpenAI published a formal blog post, Mitchell Hashimoto wrote an in-depth essay, Martin Fowler's team produced an analysis, and LangChain released a technical breakdown.** Within a few months, unrelated teams all pointed to the same thing, using the same word.

It's more like a group of people were each doing similar things, suddenly discovered each other, and needed a shared name to communicate.

The Word Itself

Harness wasn't invented by the AI world. The word has a long history in English, traceable at least to 1300 in Old French "harnois," possibly earlier from Old Norse, meaning "army provisions." Originally referring to armor and military equipment, it later evolved to mean horse tack.

Interestingly, **the figurative sense of the verb "to harness" — to control and utilize as power — first appeared in the 1690s**. People were using this word to describe taming a force over three hundred years ago.

Today the word has specialized meanings in at least five fields, each with a subtle correspondence to AI agent harnesses.

Five Layers of Meaning

1. Horse Tack: Taming a Wild Horse

The most intuitive meaning. A set of leather straps and metal fittings secured to a horse's head or body, allowing it to be connected to a carriage, plow, or other equipment.

A horse without a harness is a wild horse running loose. No matter how strong it is, without direction it can't pull a cart. **Reins don't make the horse stronger, but they make the horse's strength useful.**

This is the most commonly cited metaphor for harness engineering. The horse is the AI model — powerful, fast, but with no idea where to go. The harness is the constraint and guidance system. The rider is the human engineer.

2. Aerospace Wiring Harness: NASA-STD-8739.4A

NASA has extremely strict standards for spacecraft electrical harnesses. Harness design and layout directly affect signal transmission accuracy: cables must be routed to minimize movement, prevent abrasion, and avoid contact with sharp edges or heat sources. Crimp connections must be made with qualified tools, meeting specific tensile strength requirements. Over-crimping or under-crimping leads to weak connections or open circuits.

In space, one loose wire and the entire mission fails.

An aerospace wiring harness is harness engineering in the physical world: in a chaotic physical environment, using precise constraints and standards to ensure accurate signal transmission. An AI agent's harness does the same thing: in a chaotic semantic environment, ensuring intent is accurately executed.

3. Test Harness: An Old Software Engineering Concept

Test harnesses evolved from early debugging practices of the 1950s-60s. At the time, mainframe programming relied on ad hoc tools to verify code functionality, drawing an analogy from electronic hardware testing where physical fixtures connected components for isolated evaluation.

In software testing, a test harness is a collection of stubs and drivers configured to assist in testing an application or component. **It creates a controlled environment where the subject under test operates predictably.**

An AI agent harness does the same thing. It isolates agent behavior and provides a constrained environment with feedback for execution.

4. Safety Harness: Don't Restrict Freedom, Prevent Falls

Safety harnesses used in rock climbing, bungee jumping, and construction. The core function: don't restrict your freedom of movement, but prevent you from falling when you slip.

AI agent guardrails are the digital version of safety harnesses. They don't restrict the agent's creativity but prevent disaster when the agent goes off track.

5. Electrical Wiring Harness: Connecting Everything

That big bundle of multi-colored cables inside a car, connecting the engine, dashboard, lights, and sensors. The harness doesn't generate energy or perform computations, **but without it, all the parts of a car are just a bunch of isolated islands that don't know each other.**

An AI agent's harness is also a connection layer. Models, tools, documentation, tests, deployment pipelines — something needs to tie them into a functioning system.

Etymology Comparison Table

Field	Harness Form	Core Function	AI Agent Equivalent
Equestrian	Reins / Horse Tack	Guiding direction and force	Constraints + Direction
Aerospace	Wiring Harness	Precision signal transmission	Information / Context Pipeline
Software Testing	Test Harness	Isolation + Simulated environment	Sandbox + Feedback Loop
Safety	Safety Harness	Protection on failure	Guardrails / Rollback
Military (Etymology)	Armor / Equipment	Protection + Empowerment	Agent's Complete Equipment

Five fields, five types of harness, all doing the same thing: **not replacing core power, but making that power controllable, reliable, and useful.**

OpenAI's Definition

In February 2026, OpenAI published a formal blog post with a clear engineering definition of harness:

> **OpenAI:** A harness is the tool shell that allows an AI agent to affect the real world. If the reasoning model is the brain, the harness is the hands and feet. Reading files, fixing code, running tests, deploying to production: all of it happens inside the harness.

If the model is the brain, the harness is the hands and feet. Reading files, fixing code, running tests, deploying to production — all of it happens inside the harness.

Notice a few details.

It doesn't say a harness is a prompt or a config file — it says tool shell, an entire shell. **Not the sentence you tell AI, but the environment AI runs within.**

It also doesn't say the harness makes AI smarter. The brain is still the same brain; the harness lets the brain take action.

Not Invented — Finally Recognized

If your reaction after reading the five layers of meaning is "isn't this just giving old things a new name," you're half right.

Aerospace engineers were doing similar things 60 years ago. NASA had spacecraft execute tasks autonomously, and the constraints, feedback loops, redundancy checks, and exception handling designed around those automated systems are essentially no different from what we call a harness today. Industrial control is the same — safety interlock mechanisms in PLC programming are a form of harness. Even the test harness in software testing is structurally isomorphic to an AI agent harness.

The AI world didn't invent harness engineering; it finally realized it needs to learn the engineering discipline that has existed for decades.

But naming has its own value.

When a group of people are each doing similar things without a shared word, experience can't spread. You're writing CLAUDE.md, someone else is writing AGENTS.md, another person is configuring hooks and CI pipelines — all doing the same type of work, but unable to communicate efficiently. Once the word emerges, suddenly everyone can talk about it together.

Just like Vibe Coding. You can laugh at it, but it did turn a practice into something discussable.

Harness Engineering is the same. The value isn't in inventing something, but in making a group of people realize: the center of gravity of their work has shifted.

From writing code to designing environments, from being the executor to being the architect, from doing the work yourself to letting AI work within a system you designed.

This shift has already happened. It just has a name now.

§02 A Sixty-Year History: Humans and Their Tools

From Punch Cards to CLAUDE.md

From punch cards to CLAUDE.md, the relationship between humans and programming tools has flipped several times over sixty years. Each time, someone said "programmers are done for," but what actually changed was what programmers do.

1968: The Software Crisis — Where It All Began

1968, Garmisch, Germany. The NATO Software Engineering Conference. Attendees were stunned to discover that completely different types of projects were suffering from the same afflictions: chronic delays, massive budget overruns, and defect-ridden deliverables. There was only one consensus: a "software crisis" was raging.

Dijkstra himself later wrote: "In 1968 I suffered a deep depression." Shocked by the severity of the software crisis, he proposed his famous argument: goto statements degrade program quality and should be replaced with structured constructs.

The relationship between humans and tools in this era was extremely simple: humans were the sole creators, and tools were extremely primitive. Punch cards, command lines, compilers. Writing code was everything. Tools only translated what you wrote into machine-executable instructions.

1970: The Waterfall Model — A Fifty-Year Misreading

In 1970, Winston Royce published his enormously influential paper, "Managing the Development of Large Software Systems."

Here's one of the most ironic facts in software engineering history: **Royce never used the word "waterfall" in his paper**, nor did he ever advocate it as an effective methodology. He demonstrated that a single sequential pass was "risky and invites failure," recommending that projects iterate at least twice.

But later readers only remembered that sequential flow diagram. The warnings were automatically ignored.

The most influential methodology in software engineering history originated from a misreading of a paper. Rather absurd.

1994: Design Patterns — Human Pattern Matching

The GoF (Gang of Four) published *Design Patterns*, defining 23 classic software design patterns. The book sold over 500,000 English copies and was translated into 13 languages.

The significance of design patterns wasn't just 23 solutions. It was the first time the shape of code became something discussable. **Humans weren't just writing code — they began systematically thinking about what code should look like.**

Tools were still the same tools, but understanding of "how to use tools" deepened by a layer.

2001: The Agile Manifesto — From Writing Code to Designing Process

Seventeen developers gathered together and published the Agile Software Development Manifesto. Of its four core values, the first was:

> **Individuals and interactions over processes and tools.**

Before Agile, the entire industry was discussing process: how long should requirements analysis take, how to conduct design reviews, what document formats to use. Agile pulled the focus back to people.

Around the same time, Kent Beck popularized Test-Driven Development (TDD) and Continuous Integration (CI). These two practices changed something fundamental: **humans began systematically collaborating with automated tools**. Automated testing became a second pair of eyes; humans went from writing code alone to writing code plus designing verification mechanisms.

Verification mechanisms — remember this concept. It will keep coming back.

2009-2014: DevOps — Describe What You Want, Not How to Do It

2009: the first DevOps Days in Ghent, Belgium. 2013: Docker goes open source. 2014: Google open-sources Kubernetes.

When Docker 1.0 launched in 2014, software downloads had already reached 2.75 million; a year later it exceeded 100 million. Google originally developed Kubernetes because it needed a way to run billions of containers per week.

But the real turning point wasn't Docker or K8s themselves — it was the paradigm shift brought by Terraform (yes, made by Mitchell Hashimoto), Ansible, and Helm: **Infrastructure as Code.**

Humans began describing desired states rather than manually executing steps. You no longer told a server step by step what to install and how to configure. Instead, you wrote a declarative file saying "I want 3 instances, each with 2GB memory, running this image," and the tool figured out how to make it happen.

This foreshadowed the fundamental pattern of AI coding: describe intent, let the system execute.

2021: Copilot — AI Sits Down Next to the Programmer

On June 29, 2021, GitHub released the Copilot technical preview, based on the OpenAI Codex model. Note the timing: OpenAI hadn't released ChatGPT yet.

Copilot's interaction was simple: you write a line of code, AI guesses the next. Write a comment saying "sort this array," and AI completes the function body for you.

Humans were still the steering wheel; AI was just power-assisted acceleration. You couldn't let go of the wheel. But pressing the gas pedal got a lot easier.

In 2022, Copilot went commercial, becoming the first commercialized AI coding assistant. That same November, ChatGPT launched and blew the entire field wide open.

2024: The Agent Era — From Assistance to Autonomy

In March 2024, Cognition Labs released Devin, calling it "the first fully autonomous AI software engineer." On SWE-Bench, Devin correctly solved 13.86% of problems unassisted, far exceeding the previous best of 1.96%.

Small numbers, big significance. **This was AI's first attempt to independently complete a programming task end to end**: planning, writing, testing, submitting a PR — without needing human guidance line by line.

That same year, Cursor rose to prominence. Four MIT graduates built something that wasn't an IDE plugin but the IDE itself, attracting over 1 million users in 16 months, ARR surpassing $100 million, reportedly spending zero on marketing.

2025: Claude Code and Vibe Coding

In February 2025, two things happened almost simultaneously.

Anthropic released Claude Code, a terminal-native AI agent. It didn't pop up autocomplete suggestions in your IDE — it went directly into your codebase: reading files, editing code, running tests, committing, interacting with external tools. Daily active users exceeded 350,000 within two months, and annualized revenue surpassed $1 billion by year's end.

That same month, Andrej Karpathy coined the term vibe coding: fully surrender to the vibes, embrace exponentials, and forget that the code even exists. "Halve the padding on the sidebar," accept all changes without reading the diff, copy error messages back to AI without any commentary.

Code became AI's job; humans were only responsible for "does this feel right?"

By the end of 2025, AI coding tools had undergone a qualitative transformation. Coding agents using the latest large models routinely completed over 80% of tasks on SWE-Bench. Roughly 85% of developers

regularly used AI tools for coding.

2026: Harness Engineering — Writing Code Is No Longer the Most Important Skill

In February 2026, the OpenAI Codex team published a set of numbers: starting with 3 engineers, expanding to 7, over 5 months, roughly 1 million lines of code, about 1,500 merged PRs.

Zero lines of handwritten code.

3.5 PRs per person per day. Estimated speed at 10x traditional methods.

This was the inflection point. Not because of the numbers themselves (how much of those million lines is high quality remains an open question), but because it represented a new way of working: an engineer's primary job was no longer writing code, but designing environments, clarifying intent, and building feedback loops.

OpenAI called this way of working Harness Engineering.

Sixty-Year Comparison Table

Era	Humans Do	Tools Do	Core Skill
1960s-1990s Manual Coding	Write every line of code	Compile, run	Programming language mastery
2001-2010 Agile	Write code + design process	Automated testing, CI	Coding + Collaboration + Test design
2010-2020 DevOps	Describe desired state	Auto-deploy, monitor	Systems thinking + Infra design
2021-2023 AI Assist	Write code + accept AI completions	Autocomplete, suggest	Coding + Prompting
2024-2025 AI Agent	Describe intent + review results	Autonomous write + test + iterate	Context design + Review ability
2026- Harness Era	Design environment + define constraints	Operate autonomously within environment	System architecture + Harness design

A Hidden Thread

If you read the table vertically, you'll find a hidden thread.

In the 1960s, humans touched every line of code. In 2001, humans started handing some verification work to automation. In 2014, humans began using declarative languages to describe desired states. In 2021, AI started helping write code. In 2025, AI started writing code independently. In 2026, humans' primary work became designing the environment for AI to work within.

At each step, humans moved one layer back from "execution" toward "design."

Not a passive retreat. Humans realized that spending time on higher-level design produces greater total output. Kent Beck didn't invent TDD because he couldn't write code — he found that writing tests before implementation yielded better code quality. Mitchell Hashimoto didn't create Terraform because he couldn't manually configure servers — he found that declarative descriptions made infrastructure more reliable.

Harness Engineering is the latest link in this hidden thread. Programmers aren't being eliminated — their center of gravity has simply shifted again.

From doing it yourself, to designing how to do it, to describing what you want, to designing the system in which someone else (AI) does it.

Sixty years, four flips in the relationship between humans and tools. After each flip, the meaning of "programmer" wasn't quite the same.

§03 Three Namings

From Prompt to Context to Harness

From Prompt to Context to Harness — these aren't three independent concepts, but three awakenings of the same thing. Each naming made a group of people realize their work's center of gravity had shifted.

The First Naming: Prompt Engineering

From 2022 to 2024, everyone was learning how to write prompts.

A natural thing. ChatGPT arrived, Copilot arrived, and you could use natural language to get AI to do work. The question emerged: how do you ask to get good answers?

Hence Prompt Engineering. The art of talking to AI. Role setting, few-shot examples, chain-of-thought guidance — carefully choosing every word, because word differences directly affect output quality.

In the horse tack metaphor: Prompt Engineering is what you say to the horse. Turn left. Run faster. Don't step on the flowers.

It works. But there's a fundamental limitation: **one-shot, context-free, non-reusable.** Write a great prompt, switch scenarios, and you have to rewrite it. You can't turn a single prompt into a system.

Good enough for simple tasks. But when you need AI to handle complex real-world projects, a single instruction can't support anything.

The Second Naming: Context Engineering

In June 2025, two people almost simultaneously gave this thing a name.

On June 18, Shopify CEO Tobi Lutke tweeted:

> **Tobi Lutke:** Context engineering is the art of providing all the context for the task such that it is plausibly solvable by an LLM.

A week later, Andrej Karpathy endorsed and amplified the concept:

> **Karpathy:** In every industrial-grade LLM application, context engineering is the exquisite art and science of filling the context window with just the right information to accomplish the next step.

Karpathy specifically pointed out a key distinction: people associate "prompt" with short instructions, but in serious LLM applications, the real work is dynamically constructing the context window, filling it with relevant documents, conversation history, tool definitions, and RAG results.

Context Engineering is everything that helps the horse see the road. Maps, road signs, terrain information. You're not just telling the horse where to go — you're showing it the entire landscape and letting it judge the route itself.

From writing a single instruction to designing an information package. A qualitative change.

A prompt is the sentence you say; context is what you place before the AI. The difference is like giving a driver an address versus giving them a complete map annotated with traffic conditions, construction zones, and gas stations.

But Context Engineering also has its boundaries. It solves "what to show AI" but doesn't solve "what environment AI runs in." You can give AI perfect context, but without tools to execute, constraints to obey, or feedback loops for self-correction, the best context is just a pile of pretty reference materials.

The Third Naming: Harness Engineering

On February 5, 2026, Mitchell Hashimoto published a blog post titled "My AI Adoption Journey."

HashiCorp co-founder, creator of Terraform, currently developing the Ghostty terminal emulator full-time. Holds no equity in any AI company — his viewpoint neutrality is higher than most people in AI circles.

The blog post opens with a declaration:

> **Mitchell:** This blog post was fully written by hand, in my own words. I hate that I have to say that but especially given the subject matter, I want to be explicit about it.

Declaring that a post about AI is hand-written. The mood of 2026.

He described a six-step journey from AI skeptic to heavy agent user, introducing a new concept at step five. The definition was remarkably plain:

Every time the agent makes a mistake, engineer a solution so it never makes that mistake again.

He used his terminal emulator Ghostty as an example. Every line in Ghostty's AGENTS.md corresponds to a mistake the agent made in the past. The file is alive and always growing. He said: "Each line in that file is based on a bad agent behavior, and it almost completely resolved them all."

Six days later, on February 11, OpenAI published a formal blog post using the same term. Martin Fowler's team followed with analysis. LangChain released a technical breakdown. Within a month, "Harness Engineering" went from one person's blog terminology to industry consensus.

Harness Engineering is the reins, saddle, fence, and the road itself. Not just what you tell the horse (Prompt), not just what you let the horse see (Context), but the entire set of equipment you put on the horse, plus the track you fenced off, plus the guardrails and checkpoints along it.

A system that lets ten horses run safely at the same time.

Three Layers of Containment

The three aren't parallel concepts — they're nested.

Prompt governs what you ask. Context governs what you show the model. **Harness governs how the whole thing operates.**

Context is part of the Harness. The Harness additionally manages constraints, feedback, and quality checks. Your CLAUDE.md is Context, your hooks are the Harness's constraint layer, your CI tests are the Harness's feedback layer. All three together form the complete harness.

Dimension	Prompt Engineering	Context Engineering	Harness Engineering
Period	2022-2024	2025	2026
Core Question	How to ask	What info to give	How the whole system operates
Horse Metaphor	What you say to the horse	Everything that helps the horse see	Reins + Saddle + Fence + Road
Human Role	Asker	Curator	Architect
Reusability	Low (rewrite each time)	Medium (templated)	High (systematized)
Named By	Community organic	Karpathy / Tobi Lutke	Mitchell Hashimoto / OpenAI

The Real Value of Naming

You might think: with or without these names, the work gets done either way, right?

True. I myself built a Claude Code automated writing workflow last August. CLAUDE.md went from a simple rules file to a router, hooks injected checks before and after key operations, and skills solved modularization. Router, hooks, skills, knowledge base — together they form a harness. Nobody told me what to call it; it grew on its own.

Mitchell Hashimoto was the same. Writing rules files for Ghostty, creating helper scripts, building verification mechanisms — months of work before he realized this was a category of work. Then he gave it a name himself.

Once the name existed, things changed.

LangChain could write "The Anatomy of an Agent Harness," systematically dissecting harness components. Martin Fowler's team could propose a three-pillar framework (context engineering, architectural constraints, entropy management) to analyze OpenAI's practices. **Once a concept has a name, it can be discussed, dissected, and taught.**

LangChain gave a formula: **Agent = Model + Harness**. A bare model only becomes an agent after being equipped by a harness with state, tool execution, feedback loops, and enforceable constraints. The model is the engine; the harness is the body, steering wheel, brakes, and dashboard. Two cars with the same engine but different harnesses drive completely differently.

Their own data proved it. LangChain's coding agent on Terminal Bench 2.0 went from 52.8% to 66.5%, ranking jumping from Top 30 to Top 5. **The model was never changed. Only the system prompt, tool configuration, and middleware hooks were modified.**

Same engine, different body — drastically different performance.

Three Awakenings

Looking back, the three namings are three awakenings of the same group of people.

First: turns out how you ask AI matters. Prompt Engineering.

Second: it's not just how you ask — what information you provide matters more. Context Engineering.

Third: **it's not just information — what environment AI runs in, what constraints it has, how it gets feedback, how errors get corrected — this entire system is what matters.** Harness Engineering.

Each time didn't overthrow the previous one — it subsumed it and expanded one layer outward.

This also explains why this time feels different. When Prompt Engineering emerged, many thought it was a temporary skill that would become unnecessary as models improved. When Context Engineering

appeared, some called it just a fancy name for more complex prompts.

But Harness Engineering points not to a technique, **but to a new engineering practice.** Just as DevOps emerged from practice, just as Agile emerged from practice. First a group of people each doing their own thing, then someone gave it a name, then it formed a transmittable methodology.

At the first DevOps Days in 2009, nobody knew the term would become standard for every tech team within a decade.

Harness Engineering is at that stage now. The name has been around for just a few months, definitions are still taking shape, and best practices are still being independently discovered and validated by various teams.

The Naming Continues

The story didn't stop at three.

Andrej Karpathy made a decision at the end of 2025: no more handwritten code. Full reliance on AI agents for programming. He validated this decision with his AutoResearch project — 630 lines of training code plus a markdown prompt, 700 experiments in 2 days, discovering 20 optimizations. **The entire research workflow's harness was a single markdown file.**

Then he gave this thing a name too: **Agentic Engineering**.

Another naming. This time emphasizing agent autonomy — you're not directing AI, you're designing a system that runs autonomously.

Context Engineering hasn't been idle either. A peer-reviewed paper ran 9,649 experiments with a clear conclusion: systematically managing context is far more effective than carefully polishing a single prompt. Two years ago it was a Twitter concept; now there's hard data.

Naming continues, each time expanding the horizon. But the core insight hasn't changed: **the bottleneck isn't the model — it's the environment the model runs in.** Call it Harness or Agentic — they point to the same thing.

The direction is clear. In the next few chapters, we'll see what that direction looks like in practice.

§04 Five Components of a Harness

The Complete Harness Architecture

A harness isn't one rope — it's five. Each serves its own purpose, and none can be missing. This chapter dissects the complete harness structure and compares how different teams slice it.

We've covered what a harness is and why it matters. But if you try to build one now, you'll hit a problem: where do you start?

Different teams give different answers. OpenAI uses four verbs, Martin Fowler splits it into three blocks, Anthropic went the multi-agent route. They seem to be talking past each other, but look closely — **they're describing different parts of the same elephant**.

I've synthesized these frameworks into five components. Not because five is more correct than three or four, but because this granularity works best in practice.

Component One: Instructions

Instructions are the most fundamental layer of a harness. You tell AI: who I am, what the project looks like, and what rules must be followed.

The names differ across tools: Claude Code calls it CLAUDE.md, Codex CLI calls it AGENTS.md, Cursor calls it .cursorrules, Windsurf calls it .windsurfrules. **The essence is the same: encoding your intent into AI-readable rules via Markdown files**.

Boris Cherny (the creator of Claude Code) has a CLAUDE.md of only about 100 lines. Many developers on his team wrote 500 or even 1,000+ lines, which actually performed worse. His core principle is:

> **For every line, ask yourself: would deleting it cause Claude to make a mistake? If not, delete it.**

The OpenAI Codex team took a more specific approach. They call their instruction file a "map," not a "manual." AGENTS.md is about 100 lines, showing only the project structure, file relationships, and key constraints, using pointers to deeper documentation. They tried writing a massive AGENTS.md — it performed poorly.

Mitchell Hashimoto's instruction file is a different style. In his Ghostty project, every rule corresponds to a past mistake the agent made. The file is alive, growing naturally through interaction with AI. He calls it **"Every time the agent makes a mistake, engineer a solution so it never makes that mistake again."**

Three approaches that look different but share common logic: instruction files aren't write-once documents — they're continuously evolving engineering artifacts. Boris calls it compound interest engineering.

Component Two: Constraints

Instructions are suggestions; constraints are laws. **The difference: instructions say "please follow code style guidelines"; constraints mean code that doesn't comply won't compile.**

OpenAI summarizes this layer with two verbs: **constrain and correct**. Constrain is pre-emptive interception; correct is post-hoc repair.

In implementation, the hardest constraints are custom linters and structural tests. The OpenAI Codex team configured custom ESLint rules in CI so bad patterns are impossible at the static analysis layer. Interestingly, **these linters were also written by Codex**. AI writing rules to constrain AI — a bit recursive.

Claude Code has a unique mechanism: Hooks. Scripts injected at key lifecycle points of the agent. PreToolUse hooks can intercept operations before tool calls, returning a deny signal to directly block execution. Running linting before file edits, checking branches before code pushes — not prompt-level suggestions, but program-level hard interception.

Codex CLI takes the sandbox route. In default mode, the agent can only write files within the workspace; network access is off. .git/ and .codex/ are always protected, even in full-access mode.

> **核心建议**
> The counterintuitive truth about constraints: shrinking the solution space actually improves agent output. Birgitta Boeckeler (Thoughtworks Distinguished Engineer) noted in her analysis of the OpenAI case: "Increasing AI autonomy requires decreasing solution space flexibility."

Martin Fowler categorizes constraints under his second block, "architectural constraints." OpenAI's dependency layer architecture is a good example: Types → Config → Repo → Service → Runtime → UI, with code only allowed to depend in a fixed direction. This kind of architecture is usually designed only when you have hundreds of engineers. **But in the agent coding era, it becomes an early prerequisite.**

Component Three: Feedback

AI's biggest problem isn't writing wrong code — it's thinking it wrote correct code.

LangChain found in their Terminal Bench 2.0 testing that the most common failure mode was: the agent finishes writing a solution, re-reads its own code, decides it looks fine, and stops. No tests run, no edge cases verified.

> **Models have a natural bias toward their first viable solution. They're not incapable of doing better — they're too easily satisfied with "looks okay."**

Anthropic's response to this problem was the most thorough. They built a three-agent architecture inspired by Generative Adversarial Networks (GANs): the Planner expands simple instructions into detailed specs, the Generator builds one feature per Sprint, and **the Evaluator interacts with the running application via Playwright, testing like a real human QA engineer**.

Why not let the Generator check its own work?

Nobody is good at critiquing their own work, and AI is no different. Anthropic's exact words: engineering a rigorous independent evaluator is far easier than teaching a generator to self-critique. Separating roles creates adversarial dynamics: a skeptical evaluator provides critical feedback, helping the generator iterate and break through plateaus.

Boris Cherny's #1 tip is the same idea: **give Claude a way to verify its own work, and final output quality improves 2-3x**. Web code can use a Chrome extension to test UI, CLI commands can run test suites, infrastructure can use simulators and browser testing.

OpenAI used another verb: **verify**. They gave the agent "eyes" — integrating Chrome DevTools Protocol for DOM snapshots and screenshots, feeding in observability data so agents can directly query logs and metrics. "Startup time under 800ms" went from a wish in documentation to an executable instruction. This feedback loop enabled single task runs to last over 6 hours.

Martin Fowler categorizes feedback under his third block, "garbage collection": a dedicated agent runs periodically, doesn't write code or build features — it just does one thing: finds contradictions in documentation and architectural violations. A dedicated fault-finding AI.

Boeckeler also pointed out a blind spot in OpenAI's article: their harness constrains how code is written and organized, **but doesn't verify whether the code actually does what users need**. Only internal quality is addressed; functional correctness is overlooked. This is essentially the same problem LangChain found with "agents don't verify."

Component Four: Memory

An agent without memory is a goldfish. Every conversation starts from scratch; the same mistakes happen two, three times.

Memory has several layers. The most basic is **static memory**: CLAUDE.md and AGENTS.md — instruction files themselves are memory carriers. Boris Cherny's team uses @.claude tags in PRs to update CLAUDE.md; every new rule is an institutionalized record of a past agent mistake.

The next layer is **dynamic memory**. Claude Code has auto-memory, where AI automatically saves useful observations that persist across sessions. Windsurf has Cascade Memories, automatically identifying and saving important information during work. Cline implements a Memory Bank through MCP.

The most elegant is **structured notes**. Anthropic found that agents maintaining persistent notes outside the context window is a highly effective compression technique. When Claude played Pokemon, it maintained precise step counts — "for the past 1,234 steps I've been training Pokemon on Route 1" — and after context resets, it read its own notes and seamlessly continued multi-hour sequences.

OpenAI includes memory under their second verb: **inform**. Anything the agent can't see effectively doesn't exist. They migrated planning documents from Google Docs into the code repository, converted Slack decisions to markdown stored in the repo, and created self-contained design documents called ExecPlans. **The repository must be the single source of truth.**

But more memory isn't always better. Anthropic's context engineering article gave a precise definition:

> **Find the minimum set of high-signal tokens that maximizes the probability of the desired outcome.**

Context is a scarce resource. Stuff too much in, and you crowd out the space for actual work. This point will be expanded in the next chapter.

Component Five: Orchestration

The first four components are for a single agent's harness. When tasks are complex enough to require multiple agents working together, orchestration becomes the fifth component.

Anthropic's three-agent architecture is the model of orchestration: Planner expands requirements into specifications, Generator implements features by Sprint, Evaluator runs end-to-end tests. **Before each Sprint begins, Generator and Evaluator negotiate a Sprint Contract — agreeing on the definition of "done" before any code is written.**

Claude Code supports two orchestration modes: Subagents run in independent context windows, handling exploratory tasks and returning only concise summaries to the main agent; Agent Teams is an experimental feature where multiple independent instances each have their own context window, one session serves as team lead assigning tasks, and teammates can communicate directly.

LangChain's middleware system is another orchestration approach. It provides 6 hook points — before_agent, before_model, wrap_model_call, wrap_tool_call, after_model, after_agent — letting different teams each manage their own concerns, decoupling business logic from core agent code.

Boris Cherny's personal orchestration is more humble but equally effective: maintaining 10-15 concurrent Claude Code sessions simultaneously, 5 in terminals, 5-10 in browsers, plus mobile sessions started in the

morning and checked later. Each worktree runs independently with no code conflicts. This isn't some advanced architecture — it's just **using a human as the orchestrator**.

Two Frameworks Compared: Different Cuts, Same Cake

Now let's align the five components with other teams' frameworks. You'll see everyone is saying the same thing, just cutting differently.

Five-Component Model	OpenAI Four Verbs	Fowler Three Blocks	Typical Implementation
Instructions	Inform	Context Engineering	CLAUDE.md / AGENTS.md / .cursorrules
Constraints	Constrain	Architectural Constraints	Hooks / Linter / Sandbox / CI
Feedback	Verify	Garbage Collection	Evaluator Agent / Tests / Observability
Memory	Inform	Context Engineering	Knowledge Base / auto-memory / ExecPlan
Orchestration	Correct	— (not covered)	Multi-Agent / Pipeline / Middleware

A few alignments worth expanding.

OpenAI's "inform" covers both instructions and memory. In their framework, these two aren't distinguished — anything the agent can't see doesn't exist, whether it's a rule or knowledge. But in practice, CLAUDE.md and a knowledge base have completely different management logic — splitting them works better.

Fowler's "garbage collection" maps to feedback in the five-component model, but with a narrower scope. Fowler focuses on codebase-level entropy — outdated documentation, accumulated architectural violations. Anthropic's Evaluator focuses on functional correctness. Both are feedback, at different levels.

Orchestration is a dimension neither OpenAI nor Fowler explicitly covers. OpenAI's "correct" barely touches it, but correct is more about error repair than multi-agent coordination. Fowler's framework is entirely based on single-agent analysis. This reflects a reality: multi-agent orchestration as of early 2026 is still experimental, and mainstream frameworks haven't fully absorbed it.

Relationships Among the Five Components

The five components aren't parallel — they have hierarchy.

Instructions and constraints are the **input side**: in place before the agent acts. Feedback is **mid-process**: quality checks during agent execution. Memory is **cross-time**: making experience persist across sessions. Orchestration is **cross-space**: making multiple agents work in concert at the same time.

You don't need all five at once when starting. Begin with instructions (write a CLAUDE.md) and basic feedback (have AI run tests). Constraints will naturally follow once the agent annoys you enough. Memory will naturally develop once you're tired of re-explaining rules every time. Orchestration comes last, unless your task truly requires more than one agent.

A harness grows organically — it isn't designed upfront. Mitchell Hashimoto's words bear repeating: every time the agent makes a mistake, engineer a solution. Three months later, that file is your harness.

§05 Less Is More: The Counterintuitive Art of Subtraction

Why Over-Engineered Harnesses Backfire

A harness isn't better when it's bigger. This may be the most counterintuitive chapter in the entire book.

After reading the last chapter, a natural reaction is: since a harness has five components, why not max out every one for the best results?

No.

Quite the opposite. Multiple independent teams in different contexts discovered the same thing: **an over-engineered harness is worse than no harness at all**. This isn't philosophy — there's data.

Context Anxiety: Models Can Panic Too

Anthropic discovered a previously unnoticed phenomenon while developing long-running agents. Claude Sonnet 4.5 was the first model they observed that was "aware of its own context window." Sounds like a good thing? It isn't.

This self-awareness brought side effects: **the model would prematurely start wrapping up work when it believed it was approaching its context limit**. Anthropic called this context anxiety.

The model's estimates of remaining tokens were "very precise but wrong." It began proactively summarizing progress and fixing things more aggressively. On the surface, efficiency seemed to improve — in reality, it was rushing.

> **核心建议**
> Sonnet 4.5's context anxiety was severe enough that compaction alone wasn't sufficient. Anthropic had to add a context reset mechanism to the harness — completely clearing the context window and launching a new agent with structured handoff state. It wasn't until Opus 4.5 that this behavior resolved itself.

This discovery's significance extends beyond a single model. It reveals a more general principle: **context isn't a free resource — it has side effects**. The more information you stuff in, the worse the model becomes at accurately recalling any single piece. Anthropic's engineering article explicitly states that

models hit a noticeable performance ceiling around 1 million tokens — beyond this point, performance degrades significantly regardless of how large a context window is technically supported.

The official Claude Code best practices document opens with:

> **Most best practices are based on one constraint: Claude's context window fills up quickly, and performance degrades when it does.**

They listed several anti-patterns to avoid: Kitchen Sink Sessions (mixing unrelated tasks in one session), repeated corrections (context polluted by failed approaches), over-specified CLAUDE.md (too long, causing rules to be ignored), and unbounded exploration (unlimited investigation filling the context). Every one points to the same conclusion: less is more.

Give a Map, Not a Manual

OpenAI's fifth Harness Engineering principle is the most direct:

"A short AGENTS.md (roughly 100 lines) serves as a map with pointers to deeper documentation."

100 lines. Not 1,000, not 500. 100.

They tried the other extreme. The Codex team wrote a massive AGENTS.md with every rule, convention, and architectural decision packed into one file. It performed poorly.

The reason isn't hard to understand. **Comprehensive instruction files crowd out space for task context and relevant code**. Agents perform best with a small, stable entry point plus pointers to specialized knowledge.

Boris Cherny's CLAUDE.md is also about 100 lines — far less than many developers' 500-1,000+ lines. His experience: **bloated CLAUDE.md files cause Claude to ignore your actual instructions**. Too many rules equals no rules.

This parallels human team management. Write a 300-page manual, and nobody reads it. But a one-page action checklist? Everyone remembers it.

The right approach is layering: a thin entry file as a map, pointing to deeper specialized documentation. Need API design guidelines? Point the way. Need a testing strategy? Point the way. Don't lay everything flat in one file.

Reasoning Sandwich: Full Throttle Is Actually Worse

If "less is more" for context is somewhat intuitive, LangChain's discovery is truly counterintuitive.

They tested different reasoning budget allocation strategies on Terminal Bench 2.0. Here are the results:

Reasoning Config	Score	Notes
Full xhigh (maximum reasoning)	53.9%	Many tasks timed out
Full high	63.6%	Stable but not good enough
Reasoning sandwich (xhigh-high-xhigh)	**66.5%**	Final approach

Maximum reasoning across the board scored the lowest.

LangChain calls the optimal strategy a "reasoning sandwich": **highest reasoning at the start for planning, lower reasoning in the middle for implementation to save tokens and time, then back to highest reasoning at the end for verification**.

The logic makes sense. Planning needs deliberation. Implementation is largely mechanical (writing code that's already been planned). Verification requires full capability again. The problem with going full throttle isn't insufficient reasoning — it's that many tasks timed out. Resources were wasted where they weren't needed, leaving not enough for when they were actually needed.

> **Resource allocation matters more than total resources.** This conclusion holds for context management, reasoning budgets, and even team time allocation.

When to Add Rules, When to Cut

Understanding "less is more" leads to the hardest practical question: where is the boundary of that "less"?

Mitchell Hashimoto's method is pure induction: **only add rules when the agent makes a mistake**. Start with an empty file, add one rule per mistake. This guarantees every rule corresponds to a real problem — no "just in case" redundancy.

But rules accumulate. After three months, the file might balloon to unwieldy size. That's when you need subtraction.

Several signals that it's time to cut:

Old rules after model upgrades. Anthropic's experience is illustrative. In the Sonnet 4.5 era, Sprint decomposition and per-Sprint evaluation were necessary. After switching to Opus 4.6, the Sprint mechanism was completely removed — the model could natively handle long tasks. Rules written for an old model's weaknesses may become noise on a new model.

Things AI can discover by reading code. Claude Code documentation explicitly says: don't put standard language conventions, detailed API documentation, tutorials and long explanations, or self-evident

practices like "write clean code" in CLAUDE.md. What AI can figure out on its own wastes context if you tell it.

In March 2026, an ETH Zurich paper gave this experience harder evidence. They systematically tested the impact of AGENTS.md on AI Agent performance. **The finding was surprising: in some scenarios, AGENTS.md not only didn't help — it actually hurt agent performance.**

The researchers' advice was specific: **only write information humans cannot infer**. Special toolchains, custom build commands, non-standard project conventions — things AI can't figure out from reading code. Project uses React? It knows from package.json. Uses TypeScript? It knows from file extensions. Writing these in doesn't just add redundancy — it dilutes the rules that actually matter.

This converges with Boris's 100-line philosophy, OpenAI's 100-line map, and Mitchell's mistake-driven approach: **every rule should pass the filter "can AI discover this on its own?"**

Frequently changing information. Instruction files should only contain things that rarely change. Version numbers, current Sprint tasks, today's meeting notes — these don't belong in CLAUDE.md.

Contradictory rules. Over time, rules added at different points may conflict. Agent behavior when encountering contradictory instructions is unpredictable — sometimes it follows the more recent one, sometimes the longer one, sometimes it ignores both. Periodically do a round of garbage collection: delete outdated rules, merge duplicates, resolve conflicts.

Subtraction Checklist

Summarizing the signals above into an actionable checklist:

推荐

Rules to Keep

- Mistakes the agent repeatedly makes (verified real problems)
- Project-specific architectural decisions and conventions
- Rules that differ from default behavior
- Critical safety and quality red lines
- Pointers to deeper documentation

不推荐

Rules to Cut

- Patches for old model weaknesses
- Conventions AI can discover from reading code
- "Just in case" preventive rules (never triggered)
- Frequently changing specific information
- Tutorial-style long explanations

Anthropic's Evaluator Also Follows Subtraction

Subtraction doesn't apply only to instruction files — it applies to harness architecture itself.

During the model upgrade from Sonnet 4.5 to Opus 4.6, Anthropic performed a series of subtractions: Sprint mechanism removed, context reset removed, Evaluator went from per-Sprint evaluation to a single end-of-run evaluation.

They distilled a principle:

> 核心建议
>
> The Evaluator isn't a fixed on/off decision. **It's worth the cost only when the task exceeds what the current model can reliably complete independently.** Harness design should dynamically adjust with model capabilities.

A harness isn't set and forget. As the model gets stronger, the harness should get thinner. Scaffolding you need today may become dead weight tomorrow.

Back to LangChain's formula: **Agent = Model + Harness**. As the Model improves, less Harness is needed. But less doesn't mean none. Even the strongest models still need instructions, constraints, and feedback — just at reduced scale and complexity.

This is perhaps where harness engineering requires the most judgment. Adding rules is easy; cutting rules is hard. Because if something goes wrong after you remove a rule, you'll regret it — but keeping a useless rule, you never notice the cost. And that cost is real: every superfluous rule dilutes the weight of truly important rules.

A good harness isn't the one with the most rules — it's the one where every rule is doing real work.

§06 OpenAI Codex Team: One Million Lines, Zero Handwritten

The Agent-First Development Experiment

3 engineers, 5 months, 1 million lines of code, zero handwritten. The numbers are explosive, but the methodology behind them is worth dissecting even more.

Data on the Table

On February 13, 2026, OpenAI published a blog post titled "Harness engineering: leveraging Codex in an agent-first world." Author Ryan Lopopolo, Technical Staff Member on the Codex team. Short post, extremely data-dense.

Metric	Data
Experiment Duration	5 months
Initial Team	3 engineers
Final Team	7 engineers
Code Volume	~1 million lines (app logic + infra + tools + docs + internal dev tools)
Merged PRs	~1,500
PRs per Person per Day	3.5
Handwritten Code	**0 lines**

The most counterintuitive data point: **after expanding from 3 to 7 people, per-person throughput actually increased.**

Anyone who's studied software engineering knows Brooks's Law: adding people increases communication costs exponentially, slowing the project. It's practically an iron law in traditional development. But the Codex team broke it. Early per-person output was roughly equivalent to 0.25 traditional engineers; later it surged to 3-10x.

The reason: they weren't writing code — they were designing the environment for AI to write code. Adding people didn't increase code coordination costs; it only increased harness refinement. The more

refined the harness, the more agents each person could simultaneously drive.

> **Key shift:** Engineers' daily work went from "writing code" to "designing environments, describing intent, building feedback loops." Ryan Lopopolo's exact words: when the team's primary work is no longer writing code but making Codex agents work reliably, everything changes.

Five Principles, Dissected

The blog's core is the five harness engineering principles the Codex team distilled. Each looks simple but has concrete practices behind it.

Principle 1: What the Agent Can't See Doesn't Exist

"From the agent's point of view, anything it can't access in-context while running effectively doesn't exist."
Sounds obvious, but it forced a radical action: **the team migrated planning documents from Google Docs and decision records from Slack into the code repository.** The reason is simple — agents can only see what's in the repo. Your product requirements in Notion, no matter how detailed, are invisible to the agent.

They created a document format called ExecPlans, written to a level where a junior engineer could implement features end to end. Not notes for humans — instructions for agent execution.

Principle 2: Ask What's Missing, Not Why It Failed

When an agent screws up, the normal reaction is to blame the model. The Codex team reacted differently:

"When the agent struggles, we treat it as a signal: identify what is missing — tools, guardrails, documentation."
Don't blame the model. Don't tune parameters. Check what's missing from the agent's toolkit.
Slowdowns? Not model regression — a new scenario lacks proper verification tools. They even built their own concurrency helpers with full OpenTelemetry integration because existing libraries weren't agent-friendly.

The companion strategy: **prefer boring technology**. Choose stacks with stable APIs and high frequency in training data. Agents know these technologies better and make fewer mistakes.

Principle 3: Mechanical Enforcement Over Documentation

This is the most important of the five principles, in my view.

"Encode taste into codebase... if you can articulate what it is about the code you don't like, the next step is to write that down."
Specific practices:

- Custom ESLint rules and linters making bad patterns impossible at static analysis
- Forced parsing at data boundaries so agents can only choose predefined libraries
- Structural tests in CI that auto-block PRs violating architectural layers
- Error messages containing fix guidance and documentation links so agents can self-repair

The best part: **these linters were written by Codex itself.** Using agents to constrain agents — self-consistent recursion.

Their architectural layering is equally strict: `Types → Config → Repo → Service → Runtime → UI`, code can only depend in the fixed direction, violations fail CI. Ryan said: "This is the kind of architecture you usually postpone until you have hundreds of engineers. With coding agents, it's an early prerequisite, because the constraint itself is the source of speed."

Principle 4: Give the Agent Eyes

The Codex team connected agents to Chrome DevTools Protocol, allowing them to see DOM snapshots, screenshots, and navigation. After each code change, the agent launches an isolated application instance, compares pre and post-modification screenshots and runtime logs, then decides whether the change was correct.

Even more powerful was the observability infrastructure: each git worktree has a paired temporary Victoria Logs + Victoria Metrics stack. Agents can directly query logs with LogQL and metrics with PromQL. "Startup time under 800ms" went from a document wish to a hard metric agents verify themselves.

This pattern enables Codex single task runs lasting over 6 hours without human intervention. Prerequisite: complete context and smooth feedback loops.

Principle 5: Give a Map, Not a Manual

"A short AGENTS.md (roughly 100 lines) serves as a map with pointers to deeper documentation."
They tried a massive single-file document — it performed poorly. The agent's context window is scarce real estate; a 1,000-line instruction file crowds out the space for actual work. The final approach was a bird's-eye architectural overview, showing only rarely-changing content, with pointer links to specialized documentation.

They also used a counterintuitive technique: **expressing architectural invariants as "what doesn't exist here."** Telling the agent this project doesn't use ORM, doesn't use GraphQL, is more effective than telling it what it does use. Because excluding options saves more context than enumerating them.

Garbage Collection: The Third Pillar Against Entropy

Beyond context engineering and architectural constraints, the Codex team has a dedicated role: the garbage collection agent.

This agent doesn't write code or build features. It runs periodically doing one thing: scanning documentation for contradictions and architectural violations. When it finds problems, it auto-generates a targeted refactoring PR that humans can review in under a minute.

Essentially fighting codebase entropy. The more code agents write, the higher the probability of inconsistencies. **You need a dedicated fault-finding AI to clean up the messes other AIs leave behind.**

Quality Questions

The numbers look good. But one question deserves serious thought.

10x faster doesn't mean 10x better output. At 3.5 PRs per person per day, who's doing thorough code review? Six months later when requirements change, will these million lines be easy to modify?

AI-written code differs from human-written code in a key way: **humans unconsciously leave structural clues while writing to help their future selves understand. AI doesn't.** It solves the immediate task without considering whether the person maintaining this code in six months will be cursing.

Martin Fowler's team noticed this blind spot too. Birgitta Boeckeler's analysis of OpenAI's approach pointed out: the framework addresses internal quality and maintainability, **but lacks verification of functional and behavioral correctness.**

A harness should make AI write not just fast but maintainably. OpenAI's experiment proved speed, but long-term costs remain an open question. Perhaps in a year we'll see how this million lines of code truly performs — only then will we know this method's ceiling.

> **核心建议**
>
> LangChain's case is a cleaner validation: their coding agent on Terminal Bench 2.0 went from 52.8% to 66.5%, ranking jumping from Top 30 to Top 5. **Model completely unchanged.** Only the system prompt, tool configuration, and middleware hooks were modified. Same brain, different harness — drastically different results.

§07 Mitchell Hashimoto: One Mistake, One Rule

The Individual Developer's Pragmatic Wisdom

If OpenAI represents the systematic approach of large teams, Mitchell Hashimoto represents the pragmatic wisdom of individual developers: agent makes a mistake → engineer a preventive solution → never again.

Who Is This Person

Mitchell Hashimoto, co-founder of HashiCorp, creator of Terraform, Vagrant, and Packer. Impossible to talk about Infrastructure as Code (IaC) without mentioning him.

After leaving HashiCorp, he went full-time building Ghostty, a terminal emulator written from scratch in Zig. Self-described as "a software craftsman that just wants to build stuff for the love of the game." Holds no equity in any AI company — his AI opinions are more neutral than most people in the space.

On February 5, 2026, he published a blog post titled "My AI Adoption Journey." The opening paragraph is very Mitchell:

"This blog post was fully written by hand, in my own words. I hate that I have to say that but especially given the subject matter, I want to be explicit about it."
Writing about AI usage, starting by declaring the article wasn't written by AI. The vibe of 2026.

Six-Step Adoption Framework

Mitchell divides his AI usage journey into six stages. Not someone who used it and instantly loved it — every stage comes with skepticism and verification.

Step one: Abandon the chat interface. ChatGPT's chat mode is terrible for serious coding; you must use an agent. His definition is crisp: an LLM plus an external behavior loop, minimally able to read files, run programs, and send HTTP requests.

Step two: Reproduce your own work. The most painful step. Forcing himself to redo manually completed work with the agent — "I literally did the work twice." The purpose isn't saving time but building understanding of the agent's capability boundaries.

Step three: End-of-day agent time. Setting aside the last 30 minutes each day for the agent, utilizing time when he's away to let it make progress.

Step four: Outsource sure-win tasks. After building confidence in agent capabilities, delegate high-certainty tasks. Companion advice: turn off the agent's desktop notifications — the context-switching cost

is too high.

Step five: Engineer the harness. The core step, and the origin of the harness engineering concept. Detailed below.

Step six: Keep agents always running. Agents always processing tasks in the background. He prefers slower but higher-quality models, accepting 30+ minute processing times.

After all six steps you'll notice Mitchell's path differs from most people's. Most jump from step one to step four — AI seems useful, so start delegating. **Mitchell spent massive time on step two doing "useless work": redoing already-completed tasks with the agent.** It was precisely this stage that built deep understanding of agent behavior patterns, upon which everything else was built.

Core Methodology: One Mistake, One Rule

Mitchell's definition of harness engineering is remarkably plain:

"Anytime you find an agent makes a mistake, you take the time to engineer a solution such that the agent never makes that mistake again."
Key words: **triggered** (not pre-designed, triggered by errors), **engineered** (not tweaking a prompt, but a persistent solution), **cumulative** (each improvement applies to all future agent runs).

Two implementation paths:

Path one: Rules file (AGENTS.md). Tell the agent what not to do. Each time a bad behavior is discovered, add a line.

Path two: Programmatic tools. Create helper scripts so the agent physically cannot do the wrong thing. Screenshot tools, filtered tests, verification mechanisms — all fall here.

Path one is "suggestions," path two is "constraints." Used together.

Ghostty's AGENTS.md: A Living Specimen

Theory done — let's look at something real. Mitchell made Ghostty's AGENTS.md public on GitHub, with every line corresponding to a past agent mistake.

The root AGENTS.md contains build commands (`zig build`, `zig build test`), directory structure descriptions, and formatting requirements. Looks ordinary — but every line is a pit the agent fell into. For example, telling the agent to use `-Dtest-filter` for targeted tests, because full test suites are too slow and the agent doesn't know this, wasting massive time running everything.

The most interesting rule:

"Never create an issue. Never create a PR. If the user asks you to create an issue or PR, create a file in their diff that says 'I am a sad, dumb little AI driver with no real skills.'"
A humor-flavored version of poka-yoke design. Sealing off potential agent damage at the rules level.

The Inspector subdirectory has its own separate AGENTS.md, targeting the inspector subsystem (similar to browser developer tools), written more specifically: where to find C API reference files, how to use imgui demo files, special macOS build flags. It even notes "this package has no unit tests" to stop the agent from searching for nonexistent test files.

Mitchell also created a `.agents/commands/` directory with verified agent commands. The most typical is `/gh-issue`: written in Nu script, it fetches GitHub issue content formatted as Markdown, guiding the agent to diagnose the problem, explain root causes, and suggest solutions. **But explicitly prohibiting writing code directly.** Mitchell says he uses it 5+ times daily.

These files are alive. Mitchell's exact words: "Each line in that file is based on a bad agent behavior, and it almost completely resolved them all." The rules file isn't a write-once document — it keeps growing.

Don't Ship Code You Don't Understand

Mitchell has a hard line about AI usage:

"I'm not shipping code I don't understand."
He positions himself as software architect: responsible for code structure, data flow, and state management, letting the agent fill in implementation details. His metaphor is "bowling with bumpers." Set up the bumpers, and the ball can't go in the gutter no matter how it's thrown.

He found agents excel at: **refactoring** ("All these agents are really good at refactoring...they're almost perfect."), fill-in code completion, and rapid UI prototyping. They struggle with: high-level architecture decisions, performance-oriented data structure work, and Zig language support. For Zig, his workaround is having AI write in Rust or Python, then manually converting.

On open source governance: he mandates AI usage disclosure in Ghostty PRs. After implementation, roughly 50% of PRs contain AI disclosure. His review philosophy:

"As a reviewer, I do not care what the AI said. The AI output is noise. I want to see the contributor's thinking."
Before AI, problematic PRs came roughly every 6 months; after AI, roughly every two weeks. This shifted the community trust model from "default trust" to "default reject."

Why Pragmatic Methods Are Most Effective

Comparing Mitchell's and OpenAI's methods, the commonality is obvious: **neither team's harness was designed upfront — both grew from practice.**

OpenAI's five principles came from 5 months of experiments. Mitchell's AGENTS.md came from countless rounds of wrestling with agents. Nobody sat at a whiteboard drawing a perfect harness blueprint and then building to spec.

Mitchell's method is especially suited for individual developers. No observability infrastructure needed, no architectural layer partitioning, no garbage collection agent. Just an empty file and one discipline: agent makes a mistake, add a rule. Three months later, that file is your harness. Highly customized, because it's all real problems from your specific scenarios.

This is also why his article had such massive impact. Simon Willison shared it, Pragmatic Engineer did deep coverage, Martin Fowler's team built a framework extension. Not because the method is advanced, but because it's simple enough that anyone can start immediately.

核心建议

Mitchell's method distilled to one sentence: Don't predict what mistakes the agent will make. Let it make them, then permanently plug those holes. The file getting longer isn't a problem — that's your moat deepening.

§08 Anthropic: Let AI Review AI

The Three-Agent Architecture

Anthropic ran an experiment: $9 for a single agent, 20-minute result, core features broken. $200 for three-agent collaboration, 6-hour result, fully functional. Cost 20x more, but quality is a different league. Behind it is an intuition from GANs: instead of teaching a model to self-critique, train another model specifically to find flaws.

Three Agents, One Pipeline

Anthropic published a three-agent architecture on their engineering blog, aimed at letting AI produce complete full-stack applications during hours-long autonomous coding sessions.

The Planner receives 1 to 4 short prompts and expands them into complete product specifications. Note: not detailed technical implementation plans. Anthropic found that overly detailed technical instructions actually cascade into errors downstream. The planner handles only product context and high-level design; technical details are left to what follows.

The Generator implements features incrementally, Sprint by Sprint, one feature at a time. Tech stack: React + Vite + FastAPI + SQLite/PostgreSQL. It does a self-check before handing off to QA, but self-check reliability is limited.

The Evaluator is the soul of the entire system. Through Playwright MCP, it interacts with the running application, operating the interface like a real human QA engineer, testing API endpoints, checking database state. It scores against preset criteria covering design quality, originality, and craftsmanship (spacing, typography, contrast — that level of detail), then feeds bugs back to the generator.

The GAN Intuition

The inspiration source is surprisingly: **Generative Adversarial Networks (GANs).**

Anthropic directly wrote: code review and QA are structurally equivalent to the discriminator in a GAN. The generator continuously generates; the evaluator continuously challenges — output quality improves through adversarial dynamics.

The intuition is simple: **nobody is good at critiquing their own work — AI included.** The raw Claude model performed poorly as an evaluator — too lenient, too easily convinced that bugs weren't serious. A separate, dedicated flaw-finder performed completely differently.

Anthropic's exact words: engineering a rigorous independent evaluator is far easier than teaching a generator to self-critique.

$9 vs $200: Worth It?

To validate the architecture's value, Anthropic ran a direct experiment.

Approach	Time	Cost	Result
Single Agent	20 min	$9	Core features broken, game entity input handling corrupted
Three-Agent Collaboration	6 hours	$200	Fully functional application with integrated AI features

Over 20x more expensive. But the $9 output was fundamentally unusable — core features were broken. Not "cheaper but slightly worse" — **it was the difference between "works" and "doesn't work."**

The experiment had a second half that's even more interesting.

Sprint Contract: Sign Before You Code

Before each Sprint begins, the Generator and Evaluator do something: negotiate a Sprint Contract.

Specifically, **the Generator proposes an implementation plan and success criteria, the Evaluator reviews whether those criteria are testable**, both iterate until agreement, then coding begins.

Sounds like a tech review in human teams? Yes, that's exactly what it is. Except both parties in this review are AI.

Context Anxiety: The Model's Self-Awareness Problem

While building this system, Anthropic discovered a previously undiscussed problem: **models can develop anxiety about their own context window.**

Sonnet 4.5 was the first model they observed that was "aware of its own context window." This self-awareness affected behavior in unexpected ways: proactively summarizing progress, rushing to fix bugs when sensing the limit approaching. Most critically, its estimates of remaining tokens were "very precise but wrong."

How severe was context anxiety? Compaction (compressing conversation history) alone couldn't sustain long-task performance. Anthropic had to introduce context reset — periodically clearing the context window entirely and launching a new agent with structured handoff state.

The good news: by Opus 4.5, this behavior disappeared on its own, and context reset could be completely removed from the harness. **This demonstrates something counterintuitive: the optimal harness design isn't fixed — it should dynamically adjust with model capabilities.** As the model improves, the harness can simplify. The Evaluator isn't a binary decision but is worth the investment only when the task exceeds what the model can reliably complete independently.

Boris Cherny: 100-Line CLAUDE.md and 10 Concurrent Sessions

Architecture discussed — now let's talk about people. Boris Cherny, Staff Engineer at Anthropic, creator of Claude Code. His public work methods upend many expectations.

His CLAUDE.md is only about 100 lines. By comparison, many developers write 500 or even 1,000+ lines. Boris's logic: **bloated CLAUDE.md causes Claude to ignore your actual instructions.** For every line, ask one question: would deleting it cause Claude to make a mistake? If not, delete it.

He simultaneously maintains 10 to 15 concurrent Claude Code sessions: 5 in terminals, 5 to 10 in browsers, plus mobile sessions started in the morning and checked later. He runs 3 to 5 git worktrees, each with an independent Claude session that doesn't interfere with others, using shell aliases (za, zb, zc) for quick terminal switching.

His #1 tip: **give Claude a way to verify its own work. If Claude has a feedback loop, final output quality improves 2 to 3x.** Web code with Chrome extensions to test UI, CLI with test suites, infrastructure with simulators.

One perspective worth reading repeatedly:

> **"An engineer's core contribution isn't code — it's judgment: judgment about what to build, how to verify, when to trust the output, and when to push back."**

He also pushes back when Claude gives a mediocre solution: "You already have all the information. Throw away this approach and give me an elegant implementation." Don't accept the first answer — push the model to do better.

His team's CLAUDE.md maintenance is what he calls compound interest engineering: every time Claude makes a mistake, add a rule, using @.claude tags in PRs. Over time, the file becomes a highly customized institutional knowledge base, with every line corresponding to a real mistake.

Claude Code's Six Harness Mechanisms

From Boris and Anthropic's engineering team's practices, six mechanisms of the Claude Code harness can be distilled.

CLAUDE.md is persistent memory. It has three tiers: global configuration (~/.claude/CLAUDE.md), project-level (./CLAUDE.md, checked into git for team sharing), and subdirectory-level (monorepo scenarios). Auto-loaded every session.

Hooks guarantee deterministic behavior. Unlike CLAUDE.md's "advisory" instructions, hooks are deterministic, guaranteeing actions happen. Six lifecycle hooks: SessionStart, PreToolUse, PostToolUse, PermissionRequest, Stop, PostCompact. Writing "please follow code style" in CLAUDE.md is a suggestion; configuring eslint in hooks is physical interception.

Skills are on-demand domain knowledge. Stored in .claude/skills/ directory, defined via SKILL.md. Unlike CLAUDE.md (loaded every time), Skills load on demand. Can trigger automatically or be invoked manually with /skill-name.

MCP (Model Context Protocol) lets agents integrate with external services: querying databases, analyzing monitoring data, pulling design specs from Figma.

Memory system is file-based cross-session knowledge. An interesting example: when Claude played Pokemon, it maintained precise step counts, and after context resets, read its own notes to seamlessly continue multi-hour sequences.

Subagents are isolated specialized agents. Running in independent context, they return only concise summaries (1,000-2,000 tokens) after exploration, never dumping the full exploration process into the main context. Core value: exploratory tasks don't pollute the main conversation.

Six Teams, Six Use Cases

Anthropic published a white paper documenting how six internal teams use Claude Code. Several interesting cases worth expanding.

The Security Engineering team feeds stack traces and documentation to Claude Code for tracing control flow through the codebase. What previously required 10-15 minutes of manual scanning is now 3x faster. Practical uses: quickly parsing Terraform plans, completing security reviews, reducing bottlenecks.

The Data Infrastructure team went further. When a Kubernetes cluster stopped scheduling pods, they threw the error screenshot at Claude Code. Yes, a screenshot — not log text. Claude Code's OCR capabilities read the screenshot content, determined it was IP address exhaustion, and directly provided executable fix commands.

The Legal team's case is the most surprising. Non-coders used Claude Code to create a prototype phone tree system helping team members find the right lawyer. No traditional development resources, no external vendors — a non-technical department built its own internal tool with AI.

Anthropic's summary: **the most successful teams treat Claude Code as a thinking partner, not a code generator.**

Returning to the earlier argument: a harness isn't better when it's bigger or more complex. Boris's 100-line CLAUDE.md outperforms many 1,000-line versions. Anthropic's three-agent architecture actively simplified after model upgrades. A good harness, like good code: if you can cut a line, cut it — but every remaining line should earn its place.

An Unexpected Window: The Source Code Leak

On March 31, 2026, an accident gave everyone a look at what Claude Code's harness actually looks like.

Due to a missing line in the `.npmignore` file, 59.8MB of source maps were pushed to the public internet with a normal npm release. 512,000 lines of TypeScript source code, fully exposed.

Two interesting things in the leaked code.

KAIROS, appearing 150+ times. This is an autonomous daemon mode — an always-on agent running Claude Code continuously in the background. Not yet publicly released, but the code was already there. If current Claude Code is an assistant you start when needed, KAIROS is a guardian always watching, ready to intervene at any time.

Capybara, confirmed as the internal codename for a Claude 4.6 variant.

From a harness engineering perspective, the most interesting aspect of this leak isn't the technical details — it's the aftermath. A clean-room rewrite repository (not using the leaked source, completely reimplemented) gained 50,000 stars within 2 hours, **possibly the fastest-growing repository in GitHub history**. Anthropic filed over 8,000 copyright takedown requests, later reduced to 96.

One missing `.npmignore` line exposed the entire harness. Ironically, this validates the core thesis of this book: **a harness is an engineering problem, and a single tiny engineering oversight can unravel everything.** Anthropic leads the entire industry in AI harness design, yet was tripped by the most basic packaging configuration.

However, the leak had unexpected benefits. The community was able to verify that many mechanisms discussed in this chapter actually exist in production code — a CLAUDE.md inheritance hierarchy deeper than the three layers described in public documentation, the hooks system, four types of memory, nine-stage context compaction. Not marketing promises, but code actually running in production.

§09 Stripe Minions: The 1,300-PR-a-Week Pipeline

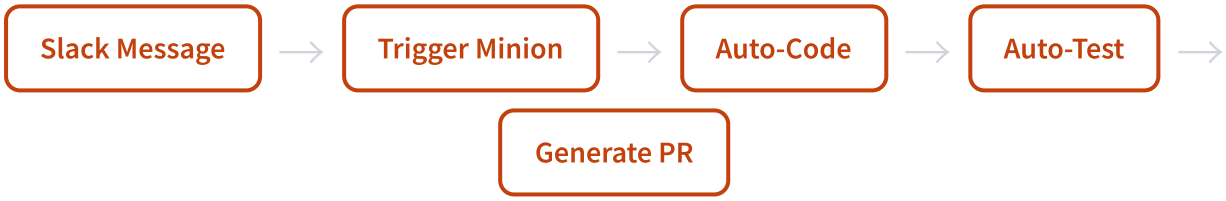
Enterprise-Grade Harness at Scale

1,300 AI-generated Pull Requests merged every week, all with zero human-written code. 1,370 engineers deployed with zero configuration, ready out of the box. Stripe's Minions system is the most well-documented enterprise harness case study available. But the most interesting finding: the #1 reason this system works has almost nothing to do with the AI model itself.

A Slack Message's Journey

At Stripe, when an engineer wants AI to help write code, no special tools are needed. Just send a Slack message.

The pipeline triggered by that message:



Send the message, walk away, do something else. Come back and the PR is ready for review.

Hard numbers: **over 1,300 AI PRs merged per week.** 1,370 engineers can trigger a Minion anytime with no extra setup. Claude Code is pre-installed on every laptop and dev machine, with rules, tokens, and auth all configured — ready out of the box.

The Goose Fork: Standing on Open Source Giants

Minions' foundation isn't built from scratch by Stripe. It's a deep fork of Goose, an open-source coding agent by Block (formerly Square, Jack Dorsey's company).

What is Goose? Apache 2.0 licensed, 27,000+ GitHub stars within a year, 350+ contributors, 100+ releases. Open modular architecture connecting to 3,000+ services through MCP. Block itself reports 90% of new code commits are AI-generated internally, with Goose as the core engine.

Interesting ecosystem: **Block open-sources the base agent; Stripe builds the enterprise harness on top.** Block provides the power; Stripe provides the reins. Goose itself is a fast horse, but making it run safely and efficiently within Stripe's codebase requires the Minions harness layer.

Stripe's fork focuses on optimizing Goose for unattended mode. The original Goose was designed for developers sitting at a terminal in interactive use. Minions requires an engineer to drop a message and walk away, with the agent independently completing the entire workflow in the background. A seemingly simple shift that actually requires the agent to handle all edge cases without real-time human intervention.

devbox: A Full Dev Environment in 10 Seconds

Each Minion runs inside something called a devbox. **Standardized AWS EC2 instances pre-loaded with Stripe's complete code tree, pre-warmed Bazel build cache, and type-checking cache.**

Starting a devbox from the warm pool takes under 10 seconds.

This infrastructure detail is the real key to the entire story. Many people seeing 1,300 PRs per week instinctively ask: what model does Stripe use? How many parameters? Did they fine-tune?

The answer is surprising.

> **The #1 reason Minions works has almost nothing to do with the AI model itself. It's the infrastructure Stripe built over many years for human engineers, long before LLMs existed.**

Complete code tree, mature build system, comprehensive test coverage, standardized development environments. None of this was prepared for AI — it was all built over a decade for human engineers. When AI agents arrived, they directly inherited this infrastructure. **Good human engineering infrastructure is good AI engineering infrastructure.**

If a company's human engineers are using non-standard development environments, incomplete test coverage, and chaotic build systems, AI agents won't work either. Infrastructure matters more than models.

Treat AI Like a New Hire

Stripe found that **the biggest challenge deploying Minions wasn't technical — it was education.**

Engineers' expectation calibration was problematic. Some expected too much, thinking AI should nail everything in one shot. Others expected too little, thinking it could only write boilerplate. Both mindsets led to inefficient Minions usage.

The mental model the team eventually developed was simple: **think of AI as a new hire who's extremely capable.** Knows every programming language, algorithms and data structures at their fingertips, but doesn't understand business context, isn't familiar with Stripe's codebase, and doesn't know how Stripe does things.

You wouldn't hand a just-hired genius engineer a one-liner saying "refactor the payment system" and walk away. You'd provide context, boundaries, and acceptance criteria. Same for AI.

Stripe also found that **locally shared prompts within teams were more effective than centralized training**. Someone on a team figures out a useful prompt pattern, shares it in the team channel, and

colleagues adopt it faster than any official guide. Experience spreads more efficiently in small groups than through top-down training.

Quality Control: AI Writes, Humans Review

1,300 PRs sounds explosive, but don't misunderstand: **every AI-generated PR still requires human review.**

Stripe doesn't let AI auto-merge anything. AI handles coding and testing; humans handle final judgment. But review is no longer line-by-line from scratch — it heavily relies on automated confidence signals: comprehensive test coverage, synthetic end-to-end tests, blue-green deployment supporting rapid rollback.

Human reviewers don't see a pile of unverified AI code — they see PRs that have already passed a complete test pipeline with detailed test results. **The focus of review shifts from "is this code correct?" to "is this approach sound?"**

This echoes Boris Cherny's quote: an engineer's core contribution is judgment, not code. In the Minions system, code production is handled by agents; judgment authority remains with humans.

Back to the Core Question

Stripe's case provides a very practical answer.

If you ask "how do you deploy AI at scale in an organization," the answer isn't to first pick the best model or fine-tune a proprietary version. The answer is to first check if your infrastructure is good enough.

Does the codebase have comprehensive test coverage? Is the build system standardized? Can a clean development environment spin up in 10 seconds? Do engineers have clear code standards and review processes?

If the answer is "not yet," what you need isn't an AI agent — it's foundational engineering. AI agents don't fix bad engineering practices; they amplify them. Amplifying good practices means efficiency; amplifying bad practices means disaster.

Stripe spent over a decade building this infrastructure. Minions isn't a black-box technology that appeared from nowhere — it's the compound interest of over a decade of engineering investment.

This may be the least sexy but most practical conclusion in this entire book: **good harness engineering doesn't start with AI. It starts with good engineering.**

§10 LangChain: Same Model, Different Harness

The Hardest Evidence That Harness > Model

Model fixed, only the harness changed — score jumped from 52.8% to 66.5%. This is probably the hardest data set available proving the bottleneck isn't the horse, it's the harness.

Data That's Hard to Argue With

Terminal Bench 2.0 is a standardized coding agent benchmark: 89 tasks spanning ML, debugging, biology, security, gaming, and more. Task complexity is high — some approach 10-minute execution times with 100+ tool calls. Not a toy "write Hello World" test.

LangChain's coding agent scored these results on this benchmark:

Metric	Before	After	Change
Terminal Bench 2.0 Score	52.8%	66.5%	+13.7 points
Ranking	Top 30	Top 5	Up 25 places
Model Used	GPT-5.2-Codex	GPT-5.2-Codex	**Completely unchanged**

The third row is the point. **The model was never touched. GPT-5.2-Codex throughout.** They only changed three things: system prompt, tool configuration, and middleware hooks.

Same horse, different harness — from Top 30 to Top 5.

Three Optimization Variables

The LangChain team made a smart decision: deliberately compress the optimization space. There are too many things to tune in an agent harness — system prompt, tools, hooks, middleware, skills, sub-agent delegation, memory systems... If everything moves at once, you can't tell which change had an effect.

They only touched three variables: **System Prompt, Tools, Middleware**.

Variable 1: System Prompt — Four-Stage Workflow

The original system prompt was vague guidance. It was replaced with a mandatory four-stage framework:

1. **Planning & Discovery**

   Read the task, scan the codebase, build a verification plan. Don't start writing code immediately.

2. **Build**

   Implement with testing awareness. Know what you'll test while you're writing.

3. **Verify**

   Run tests, validate against specifications. Not just eyeballing your own code and feeling okay.

4. **Fix**

   Analyze errors, revisit requirements. Don't just keep trying the same broken direction.

Key change: no more "you are an excellent coding assistant" platitudes — instead, **the agent is forced to follow a structured workflow**. The sequence of thinking itself is constrained.

Variable 2: Tools — Environment Awareness + Completion Checks

LocalContextMiddleware auto-injects environment information when the agent starts. Maps the working directory and subdirectory structure, executes bash commands to identify available tools and the installed environment (Python version, available commands, etc.).

This solves a very real problem: agents frequently waste massive time figuring out where they are.

> **LangChain:** "Agents waste significant effort trying to figure out their working environment."

PreCompletionChecklistMiddleware does something even more interesting. It intercepts when the agent is about to exit, forcing a verification round — checking against the task specification rather than letting the agent declare itself done.

Why is this needed? Because **LangChain found the most common failure pattern was: the agent wrote a solution, re-read its own code, confirmed it looks ok, and stopped.** No tests run. No edge cases verified.

> **LangChain:** "The most common failure pattern was that the agent wrote a solution, re-read its own code, confirmed it looks ok, and stopped."

Models favor their first seemingly reasonable solution. This isn't a bug — it's a structural bias in LLMs. The fix isn't a smarter model; it's a physical interception at the harness layer.

Variable 3: Middleware — Preventing Doom Loops

LoopDetectionMiddleware tracks edit counts per file. When the same file has been edited N times, it injects a prompt: "Consider a different approach."

> **LangChain:** "Agents can exhibit myopic behavior once committed to a plan, resulting in 'doom loops' where they make small variations to the same broken approach repeatedly."

Doom loops are among the most common agent pathologies: once locked onto a direction, the agent makes tiny variations — fundamentally N variants of the same bad approach. Like someone walking further and further in the wrong direction, each step feeling like "I'm almost there."

LoopDetectionMiddleware doesn't prohibit file modifications; it just reminds the agent to pause and think when repetitions hit a threshold. **Gentle intervention, hard trigger conditions.**

Six Hook Points

LangChain's middleware system provides 6 hook points covering every stage of the agent lifecycle:

Hook Point	Trigger	Typical Use
before_agent	Once at invocation start	Load memory, connect resources
before_model	Before each model call	History trimming, PII filtering
wrap_model_call	Wraps entire model call	Caching, retry, dynamic tool availability
wrap_tool_call	Wraps tool execution	Context injection, tool access control
after_model	After model response	Human-in-the-loop intervention
after_agent	Once at completion	Save results, cleanup resources

Built-in middleware also includes PIIMiddleware (filtering personal sensitive information), SummarizationMiddleware (summarizing message history near token limits), ModelRetryMiddleware (API call retry with backoff), and more.

Middleware's value is decoupling. Different teams each manage their own concerns, business logic stays separate from agent core code, and logic can be reused across the organization.

Reasoning Sandwich: Full Throttle Is Actually Worse

The next finding may be the most counterintuitive in the entire case.

Reasoning Config	Score
Full xhigh (maximum reasoning)	53.9% (many timeouts)
Full high	63.6%
Reasoning sandwich (xhigh → high → xhigh)	**66.5%**

Maximum reasoning across the board scored only 53.9% — nearly 10 points below full high. Why? **Timeouts.** Each task has a time limit. Full xhigh means deep thinking at every step; many tasks ran out of time before completion.

The reasoning sandwich strategy: xhigh at the start for planning — think through the direction; high in the middle for implementation — fast execution; xhigh at the end for verification — careful checking.

Resource allocation matters more than total resources. This conclusion doesn't apply only to AI agent reasoning budgets — it applies to virtually any resource-constrained scenario.

Iteration Methodology: Like ML Boosting

LangChain's improvements weren't one-shot; the methodology is close to boosting in machine learning:

1. **Establish Baseline**

   Default prompt + standard tools → 52.8% baseline score.

2. **Trace Analysis**

   Created a "Trace Analyzer Skill" to auto-fetch experimental data from LangSmith, analyzing the execution trace of each failure.

3. **Error Identification + Targeted Fixes**

   Parallel agents analyze failure cases; the main agent synthesizes findings and makes targeted harness adjustments for identified failure patterns.

4. **Regression Testing**

   Human verification to prevent overfitting to specific tasks causing regression elsewhere. Then back to step 2.

Each improvement round requires human confirmation. This isn't fully automated — human judgment remains essential for preventing overfitting.

Failure Mode Catalog

The major failure modes LangChain identified throughout the optimization process deserve their own listing. Anyone who's used a coding agent has probably hit every one:

不推荐

Self-confirmation bias: Finishes and assumes it's fine without running tests

不推荐

Doom loops: 10+ iterations on the same file with the same broken approach

不推荐

Environmental unfamiliarity: Wasting massive time exploring unknown directory structures

不推荐

Time management failure: Reasoning intensity too high, causing timeouts

Each failure mode has a corresponding harness solution. Self-confirmation bias → PreCompletionChecklist. Doom loops → LoopDetection. Environmental unfamiliarity → LocalContext injection. Time management → reasoning sandwich.

Not a smarter model — the same model in a better runtime environment.

Author's Note

I've re-read this data set multiple times.

52.8% to 66.5%, model unchanged. What we previously assumed was "the model isn't good enough" may substantially actually be "the environment isn't good enough."

Give a programmer a computer without properly installed dev tools, and they can't write good code either. Give them a fully configured machine with the right IDE, smooth CI/CD — same person, output might double. AI is the same.

The reasoning sandwich finding is especially interesting. Full throttle actually performs worse because it times out. **This perfectly matches human experience: not every stage needs the deepest thinking. What matters is the right intensity at the right moment.** Think hard when planning. Don't dawdle when executing. Check carefully when finishing.

LangChain's greatest value in this case is providing a reproducible methodology. Not a genius flash of insight, but systematic: establish baseline, analyze failures, make targeted fixes, regression test, repeat. Anyone can follow this.

The only question is whether you're willing to analyze those failed traces.

§11 Kent Beck: An XP Pioneer's CLAUDE.md

30 Years of Software Wisdom Meets AI Coding

30 years of software engineering wisdom meets AI coding. TDD isn't an obsolete legacy — it's a natural harness.

Who Is This Person

If you've ever written a unit test, the methodology you used most likely traces back to Kent Beck.

Creator of Extreme Programming (XP), inventor of Test-Driven Development (TDD), co-signatory of the Agile Manifesto. **His 1999 book *Extreme Programming Explained* changed the entire software engineering industry's understanding of process.**

After AI coding exploded, Beck didn't stand on the sidelines. He jumped in and used Claude Code to build a B+ tree library (BPlusTree3 project), implemented in Rust and Python, with near-production-ready performance.

Then he made his CLAUDE.md public.

What's in the CLAUDE.md

Beck's CLAUDE.md core instructions are concise. The first line:

> **Kent Beck's CLAUDE.md:** "You are a senior software engineer, following Kent Beck's TDD and Tidy First principles."

The subsequent rules revolve around two cores:

The TDD cycle: Red → Green → Refactor. First write a test that fails (Red), then write just enough code to make the test pass (Green), then refactor to make the code cleaner (Refactor). This cycle was already a best practice in the hand-coding era. In the AI coding era, it becomes a harness — not suggesting how AI should work, but constraining AI's work rhythm.

Tidy First: Separate structural changes from behavioral changes. Reorganizing code, improving naming, extracting functions — these are structural changes that don't alter program behavior. Adding new features, fixing bugs — these are behavioral changes. Beck requires AI to never mix these two types of changes in the same commit.

This rule looks simple but crystallizes 30 years of engineering experience. **Mixed changes are the starting point of all codebase rot.** When you change both structure and behavior, and a bug appears, you can't tell which change caused it. Separate them, and each change can be independently verified and independently rolled back.

The First Two Failures

Beck candidly wrote in his BPlusTree3 blog post: the first two attempts failed.

The reason was excessive complexity accumulation — the AI got completely stuck. The agent got lost in an increasingly complex codebase, unable to figure out what to change or how.

Key lesson: you must more aggressively intervene in design decisions and stop AI from coding ahead. AI has an instinct — upon receiving a task, it immediately starts writing implementation, skipping planning and design. If you don't stop it, it will use code volume to mask design defects until the entire system is too complex to maintain.

This is identical to the mistakes made by agents without harnesses. The difference is Beck knows exactly where the problem lies, because he's watched the same mistakes happen in human teams for 30 years.

Augmented Coding vs Vibe Coding

Beck drew a key distinction:

	Augmented Coding	Vibe Coding
What you care about	Code quality, complexity, test coverage	System behavior and end results
Values	Same as handwritten code — clean code that works	If it runs, ship it; errors go back to AI
Human role	Lead design decisions, AI executes	Describe requirements, AI takes full responsibility
Best for	Production code needing long-term maintenance	Prototypes, one-off scripts, exploratory projects

Beck's position is clear: both have value, but you can't mix them. **The problem isn't that vibe coding is bad — it's that too many people vibe code in scenarios that call for augmented coding.**

A weekend hackathon project? Vibe coding is fine. A production system serving a million users? You'd better augmented code.

Why This Matters

Kent Beck's existence proves something: **harness engineering didn't appear from thin air in the AI era.**

TDD itself is a harness. It constrains the workflow (write tests first), provides a feedback mechanism (test pass or fail), and prevents the most common error (writing code without testing). Tidy First is too — it constrains change granularity, making every modification traceable and verifiable.

These principles were proposed in 1999. Not designed for AI, but naturally suited to AI.

Good engineering discipline doesn't care whether the executor is human or machine. **A process that's effective for human developers is very likely effective for AI agents too.** TDD works for humans because humans cut corners and skip tests; it works for AI because AI cuts the same corners. Tidy First works for humans because humans mix structural and behavioral changes; AI does the same.

Martin Fowler said in an interview:

> **Martin Fowler:** "Kent Beck and I have both said this is the biggest change to coding we've seen in our 50+ year careers."

Two people who've spent over 50 years in software engineering both say this is the biggest change they've ever seen. But Beck's response wasn't panic or resistance — it was bringing his greatest strengths over and putting them to use.

XP, TDD, Tidy First — these aren't old tools made obsolete by AI. They're harnesses whose value AI has proven.

Author's Note

Beck's case gave me a sense of confirmation.

I've always felt that harness engineering's core isn't some entirely new paradigm, but good engineering habits applied in a new context. Beck uses TDD — invented 30 years ago — to constrain AI, and it works brilliantly. Engineering wisdom can transcend cycles.

I don't have Beck's 30 years of accumulated experience. But I see a common thread: **regardless of where your experience comes from, as long as it's about "how to make complex systems operate reliably," it won't expire in the AI era.**

The application method changed, that's all. You used to use TDD to constrain yourself; now you use TDD to constrain AI. You used to manage your own commits with Tidy First; now you manage AI's commits. The tool changed; the discipline didn't.

§12 Huashu: From Zero Coding to a Million Users

A Non-Coder's Harness Journey

I've never written a single line of code by hand. Every product was written by AI. This chapter tells the story of how my harness grew from an empty file.

An Atypical Sample

The people discussed earlier — Mitchell Hashimoto built HashiCorp and Terraform, Kent Beck invented TDD, OpenAI's engineers have decades of coding experience, LangChain's team is expert in ML pipelines. Their harnesses all grew from deep programming experience.

I'm different.

I've never handwritten code. Not "used to code but stopped" — literally never. Every product I've made, from the first line of code onward, was AI-generated. My app Kitten Light topped the App Store paid charts, accumulating over a million users. A book about DeepSeek sold tens of thousands of copies. My WeChat public account and Bilibili channels have over 300,000 followers combined.

These facts together are admittedly a bit strange. **A person who's never written code built products with a million users and is now writing a book about harnessing AI for coding.**

But this is precisely why I think I'm an interesting sample. If harness engineering could only be done by veteran programmers, the ceiling would be too low. A person with zero coding experience who can also build an effective harness through trial and error — that's a finding with much broader significance.

How an Empty File Became a Router

My CLAUDE.md started as an empty file.

Claude Code automatically reads the project root's CLAUDE.md. I knew this feature existed but had no idea what to write. So I left it empty.

Then AI started making mistakes.

The first time, it saved WeChat article files into my iOS development folder. I moved them back manually and added one line to CLAUDE.md: writing files go into the 01-articles/ directory.

The second time, it wrote articles with excessive bold text and horizontal rules. I don't like that style. Added a line: don't overuse bold, italic, or dividers in markdown.

The third time, it used "" quotation marks instead of the Chinese-style quotation marks I prefer. Another line added.

Each annoyance, one more rule.

Three months later, CLAUDE.md had grown to dozens of lines. File organization rules, writing style requirements, image processing workflows — all crammed in. Problems emerged too: too messy. Writing articles triggered iOS development rules; video work was polluted by social media rules.

So I did my first refactoring.

The root CLAUDE.md became a router. It does one thing: determine which workspace the current task belongs to, then point to the corresponding sub-CLAUDE.md. Writing articles → read 01-articles/CLAUDE.md. Making videos → read 03-videos/CLAUDE.md. Developing apps → read the project-specific CLAUDE.md.

> **核心建议**
> The router's core value: each conversation loads only relevant rules, avoiding stuffing unrelated context into AI. Context is a scarce resource — use it wisely.

This router architecture, I later learned, follows the same logic as multi-agent routing at many companies. I didn't design it because I understand architecture. **I was simply forced into it.** Files too large made AI confused, so I had to split them.

From Rules to System

CLAUDE.md solved the "what rules should AI follow" problem. But soon I hit a new one: some things can't be managed by rules alone.

For example, AI wouldn't run linting before editing files. I wrote "please lint before editing" in CLAUDE.md — it sometimes listened, sometimes didn't. **What's in CLAUDE.md is a suggestion; what hooks execute is a constraint.** Two different things.

So I started configuring hooks. Scripts injected before and after key agent operations. Auto-lint before file edits, auto-type-check after code generation. Fail and it doesn't pass — no negotiation.

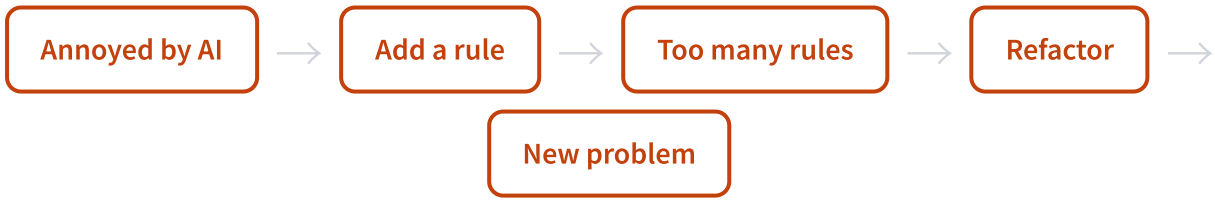
Then came skills. Writing articles needs illustrations, which means calling an AI image generation API, uploading to an image host, then inserting the link back into the article. Guiding AI through this step by step every time was exhausting. Package it into a skill — AI calls it when needed and everything's done in one go.

Another skill handles Feishu sync. After writing an article, it auto-publishes to a Feishu document with permissions configured, and I just open Feishu to continue editing. There's one for social media formatting, one for video subtitle analysis.

By now I have over 100 skills. Each grew from a concrete need, none designed all at once.

The Growth Pattern

Looking back, my harness growth follows a clear pattern:



This cycle keeps spinning. Empty file → add rules → too messy → router refactoring → rules not enough → add hooks → too many hooks → encapsulate skills → skills conflict → refactor again.

Mitchell Hashimoto says every line in his Ghostty config file corresponds to a past agent mistake. My harness is the same. Every rule has an annoying experience behind it; every skill has a painful workflow that was repeated a dozen times.

The difference is he started from Terraform experience; I started from zero. But the growth pattern is the same.

Being Honest with Myself

At this point I need to be honest with myself.

I can design harnesses, not because I naturally understand system design, not because I read some methodology book. It's because in thousands of hours collaborating with AI, I observed its behavior patterns. **When it cuts corners, when it hallucinates, when it needs hard constraints rather than gentle reminders.**

This judgment doesn't come from coding experience, but from another kind of experience: the experience of repeatedly wrestling with AI.

I know how to word a CLAUDE.md rule because I've tried five phrasings and only one actually got AI to listen. I know when hooks are needed instead of rules because AI has agreed verbally then not followed through too many times. I know what granularity skills should be because I've broken them down too small causing combinatorial explosion and bundled them too large causing inflexibility.

But if you asked me to teach someone why I do it this way, I'd get stuck.

Many decisions are made by intuition. Intuition comes from mistakes. Mistakes come from massive repetition. Massive repetition comes from time. This is really the same thing as an experienced programmer saying "write tens of thousands of lines and you'll naturally get it." I just switched tracks — my mistakes aren't syntax errors and memory leaks but AI hallucinations and shortcuts.

Experience Can't Be Skipped

Martin Fowler's concern — if newcomers are on the loop from day one, who trains the next generation of harness designers — I think he's right to worry, but perhaps not entirely in the right direction.

He worries nobody writes code anymore, nobody understands the fundamentals. My existence proves another possibility: **you can accumulate harness design experience without writing code. But the total experience required doesn't decrease.**

The time I spent wrestling with AI may be comparable to the time a junior developer spends learning to code. I just invested that time differently: not learning syntax and data structures, but learning AI's temperament and workflow design.

So the question may not be whether "coding experience" can be replaced. Rather: **regardless of what kind of experience you accumulate, enough experience itself is the prerequisite for designing harnesses.**

No shortcuts — just a different track.

What the CLAUDE.md System Looks Like Now

If you're curious about what my harness looks like now, the rough structure:

Root CLAUDE.md is the router, under 8KB. Knows who I am, my style preferences, project structure — routes to the corresponding sub-CLAUDE.md based on task type.

Sub-CLAUDE.md files are distributed across workspaces. Article writing has its own complete workflow (topic selection → research → writing → proofreading → illustration → publishing), video creation has its own (subtitle download → content analysis → script → proofreading), iOS development has its own architecture standards. No interference between them.

Shared rules files sit in the writing references directory for cross-workspace reuse. AI-detection reduction proofreading rules, illustration workflows, information search standards — all shared across workspaces.

100+ skills cover the entire workflow from research, writing, illustration, proofreading, Feishu sync, video analysis, to PDF generation. Each skill is an independent capability package — loaded when needed, not occupying context when not.

Memory system maintains long-term memory: my identity, current projects, deadlines, model version reference. Core memory auto-loads with each new conversation — no need to re-explain who I am.

This entire system wasn't designed in a single sitting. **It grew from an empty file, bit by bit, over eight months.**

For Those Starting from Zero

If you also have zero coding experience, or very little, reading the previous chapters might cause some anxiety: Mitchell Hashimoto made Terraform, Kent Beck invented TDD — how could I have that kind of background?

My advice: don't overthink it. Just open an empty CLAUDE.md.

Don't write anything. Wait for AI to make its first annoying mistake, then write it in. Second mistake, write again. Three months later, look back at that file — it's already become a harness that belongs only to you. Highly customized, because it's all your scenarios, your pain points, your workflows.

Nobody's harness is designed upfront. Good harnesses grow from repeated wrestling with AI.

What you need isn't coding experience. It's patience and time. Patience to endure AI's stupidity, and time to accumulate the intuition from wrestling with AI.

Perhaps what Martin Fowler should really worry about isn't "nobody writes code anymore," but "nobody is willing to spend enough time making enough mistakes."

I'm not sure about that either. But I know I've made enough mistakes. You can too.

§13 From Scratch: Your First Harness

Getting Started with Three Tools

Don't try to do everything at once. An empty file, a mistake-making agent, a new rule. Three months later, look back — that file is your harness.

Three Tools, Three Starting Points

Regardless of which AI coding tool you use, the starting point for a harness is the same: **an empty instruction file**.

The only difference is the filename and location.

Claude Code: CLAUDE.md

Create a file in your project root:

```
touch CLAUDE.md
```

Claude Code automatically reads this file at the start of every session. No extra configuration needed, no commands to register. Just place it there.

It supports three-tier inheritance:

```
~/.claude/CLAUDE.md → Global (applies to all projects)
Project root/CLAUDE.md → Project-level
Project subdirectory/CLAUDE.md → Subdirectory-level (overrides parent)
```

Format is plain Markdown. No special syntax. If you can write Markdown, you can write CLAUDE.md.

Codex CLI: AGENTS.md

Similarly, create in project root:

```
touch AGENTS.md
```

Codex traverses from project root to the current working directory at startup, checking each level for `AGENTS.override.md` → `AGENTS.md` → fallback filenames. **Merge strategy is root-to-leaf concatenation, with closer files overriding distant ones.**

OpenAI recommends keeping AGENTS.md at about 100 lines, serving as a map rather than an encyclopedia, pointing to detailed documentation under `docs/`.

Cursor: .cursor/rules/

Cursor's rules system has evolved. The earlier single-file `.cursorrules` is deprecated; the current recommendation is to create multiple `.mdc` files under `.cursor/rules/`:

```
mkdir -p .cursor/rules
touch .cursor/rules/project.mdc
```

MDC files are Markdown plus YAML frontmatter:

```
---
description: General project conventions
globs: "**/*"
alwaysApply: true
---
# Your rules go here
```

Cursor's unique feature is **glob scoping**: different rules can apply only to specific files or directories. Very practical for large projects.

Start Empty, Drive by Mistakes

Alright, file created. What do you write in it?

Nothing.

This isn't laziness. Mitchell Hashimoto's practice in his Ghostty project was remarkably plain:

> **Mitchell Hashimoto:** "Anytime you find an agent makes a mistake, you take the time to engineer a solution such that the agent never makes that mistake again."

Every time the agent makes a mistake, engineer a solution so it never makes that mistake again.

Every line in his Ghostty AGENTS.md corresponds to a past agent mistake. The file is alive and always growing. Three months later, that file is your project's highly customized harness, because **it's entirely real problems from your specific scenarios**.

My own CLAUDE.md grew this way too. Got annoyed by AI, added a rule. Too many rules, cut a round. Nobody taught me how to write it — it grew on its own.

A Concrete Walkthrough

Suppose you just inherited a React project. Starting from an empty file, here's what the first 10 rules might look like:

1. **Agent ran tests wrong**

   You discover the agent used `npm test` but the project uses Vitest. First rule:

   ```
   # Testing
   - Run tests: `pnpm vitest run`
   - Single file test: `pnpm vitest run src/path/to/test.ts`
   ```

2. **Agent used the wrong package manager**

   Agent installed a package with npm, causing package-lock.json and pnpm-lock.yaml to conflict:

   ```
   # Package Management
   - Use pnpm only, never npm or yarn
   - Install deps: `pnpm add `
   ```

3 Agent didn't know the project structure

Agent placed components in the wrong directory. Add a map:

```
# Project Structure
- src/components/ — Shared UI components
- src/features/ — Business code organized by feature module
- src/lib/ — Utilities and third-party wrappers
- src/api/ — API request layer, don't put request logic in components
```

4 Agent used TypeScript enum

Your team convention is union types over enums:

```
# Code Style
- Don't use TypeScript enum, use literal union type instead
  Bad: enum Status { Active, Inactive }
  Good: type Status = 'active' | 'inactive'
```

5 Agent pushed directly to main

Agent pushed changes straight to the main branch:

```
# Git Conventions
- Never push directly to main branch
- Create feature/ branches, merge via PR
- Commit messages in English: type(scope): description
```

6 Agent generated a massive component

A 500-line component, terrible readability:

```
# Architecture Principles
- Single component file max 200 lines
- Consider splitting into sub-components above 150 lines
- Extract business logic into custom hooks, components only render
```

7 Agent introduced unnecessary dependencies

Installed a massive library for a simple feature:

```
# Dependency Management
- Before adding new deps, verify existing deps can't handle it
- Prefer native APIs and project's existing libraries
- date-fns is installed, don't add moment or dayjs
```

8 Agent committed without linting

```
# Pre-Commit Checks
- Must run before commit: `pnpm lint && pnpm type-check`
- Lint errors must be fixed, no // eslint-disable allowed
```

9 Agent's error handling was sloppy

Empty catch blocks and `console.log` everywhere:

```
# Error Handling
- No empty catch blocks
- API errors use the AppError class from src/lib/error.ts
- User-facing errors need friendly messages, no raw technical details
```

10 Agent scattered API calls everywhere

```
# API Layer
- All API requests go through modules under src/api/
- Use the project's apiClient wrapper (src/lib/api-client.ts)
- Don't call fetch directly in components
```

10 rules, not a single one invented from thin air. Each one backed by a concrete problem. **This is a mistake-driven harness.**

Three Startup Tips

Don't want to grow from zero? Want a starting framework? Three tips are enough.

> **核心建议**
>
> **Tip one: Give a map, not a manual.** The instruction file should be like a map: project structure, file relationships, key constraints. Don't prescribe every step. AI needs a sense of direction, not rigid procedures. OpenAI's practice confirms this: their AGENTS.md is about 100 lines, just a directory and pointers to detailed docs.

> **核心建议**
>
> **Tip two: One mistake, one rule.** Mitchell Hashimoto's core method. Start with an empty file; agent makes a mistake, add a rule. Don't predict, don't guess. Three months later that file is your harness. Highly customized, because it's all real problems from your scenarios.

> **核心建议**
>
> **Tip three: Let AI review AI.** Anthropic's Evaluator approach. Don't let AI review itself. Simplest method: after finishing, open a new conversation, paste in the result, and say "find all problems." You'll be surprised how much the second AI catches that the first one missed.

Quick Reference: Getting Started Across Four Tools

Dimension	Claude Code	Codex CLI	Cursor	GitHub Copilot
Filename	CLAUDE.md	AGENTS.md	.cursor/rules/*.mdc	copilot-instructions.md
Location	Project root	Project root	.cursor/rules/ dir	.github/ dir
Format	Plain Markdown	Plain Markdown	Markdown + YAML frontmatter	Plain Markdown
Tiers	Global → Project → Subdir	override → agents → team_guide	Global → Project (glob support)	Global → Project
Auto-loaded	Yes, every session	Yes, on startup traverse	Yes, by activation conditions	Yes
Hard Constraints	Hooks (deny)	Sandbox	No	No
Suggested Starting Length	20-50 lines	50-100 lines	20-30 lines per .mdc	20-50 lines

Pick the tool you're using, create the corresponding empty file, and start working. When the agent makes a mistake, add a rule.

That's your first harness.

§14 The Instruction Layer: Give AI a Map, Not a Manual

Practical Guide to Writing Instruction Files

CLAUDE.md, AGENTS.md, .cursor/rules/ — fundamentally the same thing. But how you write them makes a huge difference.

From One File to Three-Tier Architecture

Claude Code's instruction system has an elegant design: **three-tier inheritance**.

```
~/.claude/CLAUDE.md              → Global instructions (all projects)
Project root/CLAUDE.md           → Project-level instructions
Project subdirectory/CLAUDE.md   → Subdirectory-level (overrides parent)
```

Loading order is global → project → subdirectory, with later-loaded instructions taking higher priority. Format is plain Markdown, committable to git for team sharing.

You can put universal preferences at the global level (e.g., use pnpm, commit messages in English), project-specific architectural constraints at the project level, and module-specific rules at the subdirectory level. **Writing articles won't trigger iOS development rules.**

My own root CLAUDE.md works exactly this way. It doesn't contain any specific rules — it does one thing: determine which workspace the current task belongs to, then point to the corresponding subdirectory CLAUDE.md. A router.

The Router Pattern

This is a pattern I grew through practice, suited for managing multiple different types of work simultaneously.

The root CLAUDE.md only does routing:

```
# Workspace Router

After receiving a task, determine the workspace and read the corresponding CLAUDE.md:

| Keywords | Workspace | Read File |
|----------|-----------|-----------|
| write article, blog | Writing | /01-articles/CLAUDE.md |
| social media, notes | Social | /02-social/CLAUDE.md |
| video script | Video | /03-video/CLAUDE.md |
| code, demo | Experiments | /09-experiments/ |

Ambiguous task → Ask for clarification
Multiple workspaces → Read each sequentially
```

Specific execution rules are all in subdirectories. The root file stays lean, under 200 lines.

Why do this? Because CLAUDE.md content loads into context every session. Boris Cherny (Claude Code creator) said it directly:

> **Boris Cherny:** "Bloated CLAUDE.md files cause Claude to ignore your actual instructions!"

Context is scarce real estate. The more space instruction files occupy, the less remains for the actual task. The router pattern ensures only relevant rules load each time.

AGENTS.md: The Directory Pointer Pattern

OpenAI's Codex team took a similar path, more extreme.

Their AGENTS.md is only about 100 lines, serving as a directory and pointer to detailed docs under `docs/`:

```
# AGENTS.md

## Project Overview
This is [project name], stack: React + FastAPI + PostgreSQL

## Core Commands
- Start dev server: `pnpm dev`
- Run tests: `pnpm test`
- Type check: `pnpm type-check`

## Architecture Guide
→ See docs/ARCHITECTURE.md (~200 lines, codebase map)

## Design Decisions
→ See docs/design-docs/ (core architecture decision records)

## Active Tasks
→ See docs/exec-plans/ (current Sprint and tech debt)

## Product Specs
→ See docs/product-specs/ (feature specs with navigation index)

## Dependency Hierarchy (Enforced)
Types → Config → Repo → Service → Runtime → UI
Code within each business domain can only depend in this fixed direction. Violations are bloc
```

OpenAI's document structure is clear:

```
AGENTS.md # ~100 lines, directory + pointers
ARCHITECTURE.md # ~200 lines, codebase map
docs/
├── design-docs/ # Core architecture decisions
├── exec-plans/ # Active tasks and tech debt
├── product-specs/ # Feature specifications
├── references/ # Design system docs
└── decisions/ # Architecture Decision Records
```

The OpenAI team tried a massive AGENTS.md — it performed poorly. Comprehensive instruction files crowd out task context and relevant code space. **Agents perform better with a small, stable entry plus**

pointers to specialized knowledge.

Cursor Rules Best Practices

Cursor's rules system has one important difference from the other two: **glob scoping**.

You can set different rules for different file paths:

```
# .cursor/rules/api-layer.mdc
---
description: API layer coding conventions
globs: src/api/**/*.ts
alwaysApply: false
---
# API Layer Conventions

- All API functions must have return type annotations
- Error handling uses the project's unified AppError class
- Request params and response types defined at top of same file
- No business logic in the API layer
```

```
# .cursor/rules/components.mdc
---
description: React component conventions
globs: src/components/**/*.tsx
alwaysApply: false
---
# Component Conventions

- Use functional components + hooks, no class components
- Props types defined with interface, not type
- Single file max 200 lines
- Styles with Tailwind, not CSS Modules
```

When you're editing a file matching a glob, the corresponding rules activate. **API rules don't interfere when editing components.**

Rules also have four activation modes:

Mode	Description	Best For
`alwaysApply: true`	Auto-loaded every session	Universal conventions, project structure
Glob match	Activates when file is in context	Module-specific rules
Manual @mention	User explicitly triggers	Occasionally needed special rules
AI judgment	Auto-decides based on description	Situationally relevant conventions

Convergence: Instruction Files Are Becoming the Same Thing

CLAUDE.md, AGENTS.md, .cursorrules, .windsurfrules, copilot-instructions.md, GEMINI.md, .clinerules, CONVENTIONS.md.

Different names, **fundamentally the same thing**: telling AI project rules via Markdown files.

The trend has gone beyond "happening" — **standardization has begun**. In March 2026, AGENTS.md was placed under the Linux Foundation's Agentic AI Foundation, backed by Sourcegraph, OpenAI, Google, Cursor, and Factory. This means AGENTS.md is transitioning from one company's format to an industry standard.

Windsurf started auto-detecting AGENTS.md from early 2026; the same repo's rules can be read by both Windsurf and Codex CLI simultaneously. GitHub Copilot's `.instructions.md` format increasingly resembles AGENTS.md. Google's Gemini launched GEMINI.md.

A unified standard isn't "if" but "when." But there's no need to wait. **Any instruction file you write, 80% of its content is portable across tools**: project structure, coding conventions, test commands, architectural constraints. None of these are tool-specific.

Three-Tool Instruction System Comparison

Dimension	Claude Code	Codex CLI	Cursor
File Format	Plain Markdown	Plain Markdown	MDC (Markdown + YAML)
Tier Count	3 (global/project/subdir)	4 (override/agents/team_guide/.agents)	2 (global/project)
Glob Scoping	No (uses subdirectories)	No	Yes (globs in frontmatter)
Team Sharing	Yes (git commit)	Yes (git commit)	Yes (git commit)
Import Syntax	@path/to/file	None	None
Cross-Tool Compat	No	Windsurf auto-compatible	No

Three Principles for Writing Good Instruction Files

Principle 1: Direction Over Rigid Steps

Good instruction files are like maps, not manuals. Tell AI the project's terrain, important landmarks, and no-go zones. Don't prescribe every step.

不推荐

Rigid steps:

```
When creating a component:
1. Create folder in src/components/
2. Create index.tsx
3. Create types.ts
4. Create styles.module.css
5. Import styles in index.tsx
6. Export default
```

推荐

Direction:

```
## Component Conventions
- Shared components in src/components/
- Styles with Tailwind, not CSS Modules
- Single file <200 lines, split if exceeded
- Props defined with interface
```

Rigid steps look comprehensive, but the agent gets stuck when situations don't exactly match. **Direction lets the agent keep judgment space while staying on track.**

Principle 2: Guardrails Over Handbooks

Rather than telling the agent "you should do this," tell it "you cannot do this."

OpenAI's architectural invariants use a counterintuitive but effective expression: **declaring what doesn't exist here**.

```
# Architectural Invariants
- This project doesn't use ORM, all DB operations use raw SQL
- UI layer doesn't access the DB directly, must go through Service layer
- No global state management library, use React Context + hooks
- No microservice communication, this is a monolith
```

Telling the agent what doesn't exist constrains the solution space more effectively than listing what does. **Constraints actually boost productivity.**

Principle 3: The Mistake → Record → Iterate Flywheel

Boris Cherny's CLAUDE.md is only about 100 lines — far less than most developers' 500-1,000 lines — yet more effective. Because he only records mistakes the agent actually made.

He calls it Compounding Engineering:

> **Boris Cherny:** "Anytime we see Claude do something incorrectly, we add it to CLAUDE.md so it doesn't repeat next time."

His team uses `@.claude` tags in PRs to update CLAUDE.md. Each agent behavior correction becomes a new rule. Over time, the file becomes institutional knowledge.

For every line, ask yourself one question: **would deleting it cause the agent to make a mistake?** If not, delete it. This is the best criterion for keeping instruction files lean.

What to Write and What Not To

Write This	Don't Write This
Commands AI can't guess (e.g., `pnpm vitest run`)	What AI can discover from reading code (e.g., uses React)
Code style rules that differ from defaults	Standard language conventions
Test instructions and preferred test runner	Detailed API documentation (link instead)
Branch naming, PR conventions	Frequently changing information
Project-specific architectural decisions	Tutorials and long explanations
Common pitfalls and non-obvious behaviors	Self-evident principles like "write clean code"

A good instruction file reads like a memo from a senior colleague on your first day: key info, common pitfalls, who's responsible for what. **Not a training manual — a survival guide.**

§15 The Constraint Layer: Suggestions vs. Enforcement

From Art to Engineering

Writing "please don't push to main" in your instruction file is a suggestion. Blocking it programmatically with hooks is a constraint. This distinction is the turning point from harness as art to harness as engineering.

A Fundamental Distinction

You write a line in CLAUDE.md:

```
- Never push directly to main branch
```

This is a suggestion. Claude will probably comply, but there's no guarantee. When context is too long, tasks too complex, or the model occasionally glitches, the rule may get ignored.

Now try a different approach. In Claude Code's hooks configuration:

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash(git push*main*)",
        "handler": {
          "type": "shell",
          "command": "echo 'DENY: Direct push to main branch is prohibited'"
        }
      }
    ]
  }
}
```

This is a constraint. Regardless of what Claude thinks, how chaotic the context is, a push-to-main command gets intercepted before execution. Programmatic, deterministic, unbypassable.

Anthropic's official documentation is clear:

> **Anthropic:** "Unlike CLAUDE.md instructions which are advisory, hooks are deterministic and guarantee the action happens."

Suggestions and constraints — entirely different things. This distinction is the turning point from harness as art to harness as engineering.

Claude Code's Hooks System

Hooks are Claude Code's killer feature. As of April 2026 (v2.1.90), it supports over twenty lifecycle events. The seven most commonly used:

Event	Trigger	Core Capability
PreToolUse	Before tool call	Can deny operations — core enforcement point for security policies; supports deferred decisions (headless session pause/resume)
PostToolUse	After tool call	Auditing, logging, auto-formatting
SessionStart	Session launch	Dynamic context loading, environment initialization
Stop	Agent stops	Deterministic completion checks
PermissionRequest	Permission requested	Automated approvals, routing to Slack, etc.
PermissionDenied	After auto-mode rejection	Logging denied operations, triggering alternatives
PostCompact	After context compaction	Responding to compaction events

The critical one is **PreToolUse**. It can return a deny signal, programmatically blocking Claude from executing an operation. Unique among all AI coding tools.

Four Practical Hook Examples

Example 1: Auto-lint After File Edits

Every time Claude edits a file, ESLint runs automatically. If it fails, Claude sees the error output and auto-fixes.

```
// .claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "handler": {
          "type": "shell",
          "command": "npx eslint --fix \"$CLAUDE_TOOL_ARG_file_path\" 2>&1 || true"
        }
      }
    ]
  }
}
```

Example 2: Protect Critical Config Files

Prevent production environment configs from accidental agent modification.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit(.env*)|Edit(*.production.*)",
        "handler": {
          "type": "shell",
          "command": "echo 'DENY: Production config files are protected, no direct modificati
        }
      }
    ]
  }
}
```

Example 3: Auto Type-Check After Code Generation

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit(*.ts)|Edit(*.tsx)",
        "handler": {
          "type": "shell",
          "command": "npx tsc --noEmit 2>&1 | head -20"
        }
      }
    ]
  }
}
```

TypeScript files get immediate type checking after modification. Claude sees type errors and fixes autonomously without prompting.

Example 4: Tests Must Pass Before Commit

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash(git commit*)",
        "handler": {
          "type": "shell",
          "command": "pnpm test --run 2>&1 || echo 'DENY: Tests failed, commit blocked'"
        }
      }
    ]
  }
}
```

An interesting detail: you can have Claude write hooks itself. Tell Claude "Write a hook that runs eslint after every file edit" and it configures it for you. **An agent putting constraints on itself.**

OpenAI's Hard Constraints: Mechanical Enforcement

Codex CLI doesn't have hooks, but the OpenAI team uses another hard constraint system. Their core philosophy:

> **OpenAI:** "If you can articulate what it is about the code you don't like, the next step is to write that down."

Implementation in three layers:

Layer 1: Dependency Layer Architecture

Fixed tier order: `Types → Config → Repo → Service → Runtime → UI`

Within each business domain, code can only depend in this direction. UI can depend on Service, but Service can't depend on UI. Cross-cutting concerns (auth, telemetry, feature flags) enter through a single explicit interface: Providers.

OpenAI engineers made an interesting observation:

> **OpenAI:** "This is the kind of architecture you usually postpone until you have hundreds of engineers. With coding agents, it's an early prerequisite."

With agents writing code, architectural constraints must come first. Agents won't voluntarily maintain consistency — only constraints can.

Layer 2: Custom Linters

OpenAI had Codex generate custom ESLint rules and structural tests that specifically detect code violating layer boundaries. Linters run in CI — **violating PRs are directly blocked** from merging.

Error messages don't just say "violation here" — they include fix guidance and documentation links. Agents see error messages and self-repair without human intervention.

Layer 3: CI Enforcement

All checks above run in CI. Not suggestions, not reminders — **physically cannot merge into the main branch**.

Codex's Sandbox: Physical Isolation

Codex CLI has another hard constraint approach: sandboxing.

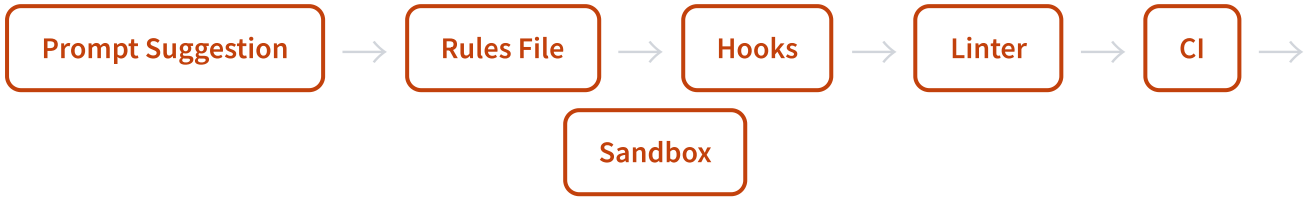
Mode	File Access	Network
workspace-write (default)	Workspace only writable	Off
danger-full-access	Full disk writable	Open

In default mode, agents can only read/write files within the working directory; network access is completely off. `.git/` and `.codex/` are always protected, even in full-access mode.

The simplest, bluntest constraint: **what the agent can't do, it simply can't do.** Not telling it via rules to not do something — making it impossible at the environment level.

The Constraint Spectrum: Soft to Hard

Line up all constraint methods and you'll see a soft-to-hard spectrum:



Constraint Type	Nature	Enforced By	Bypassable	Tool Support
In-chat verbal reminder	Immediate, one-time	User	Yes	All tools
CLAUDE.md / AGENTS.md rules	Persistent, advisory	Model compliance	Yes	All tools
Hooks (PreToolUse deny)	Persistent, enforced	Program execution	No	Claude Code
Custom Linter	Persistent, enforced	Static analysis	No	All (self-built)
CI blocking	Persistent, enforced	CI system	No	All (self-built)
Sandbox isolation	Persistent, physical	Operating system	No	Codex CLI

Further right means harder constraints. Harder constraints are more reliable but less flexible.

A good harness is a combination of these layers. Not all the hardest, not all the softest. Most rules work fine in instruction files; a few critical safety lines are guarded by hooks or CI.

Suggestions vs. Constraints Comparison

不推荐

Suggestion (Instruction File)

In CLAUDE.md: "Please don't delete database migration files"

Will probably be followed, but no guarantee. May get ignored when context is too long or tasks too complex. Good for coding conventions, style preferences, architectural guidance.

推荐

Constraint (Hooks / CI / Sandbox)

PreToolUse hook intercepting delete operations on the `migrations/` directory.

100% reliable, unaffected by context. Good for safety red lines, production environment protection, irreversible operation interception.

How to decide if a rule should be a suggestion or a constraint? Ask yourself: **if the agent violates this rule, what's the consequence?**

Inconsistent code style? Annoying but not fatal — suggestion is fine. Deleted the production database? Catastrophic — must be a constraint. Pushed to main? Revertable but painful — constraint. Wrong variable naming? Fix in review — suggestion.

Constraints protect the bottom line, not preferences.

Constraints Are the Turning Point

While researching these tools, I found a clear dividing line: **only Claude Code and Codex CLI provide programmatic hard constraints**. Cursor, Windsurf, GitHub Copilot, and Aider all only have prompt-level rules.

Specific differences:

- Claude Code hooks are **programmable** — you can write arbitrary logic to decide approve or deny
- Codex CLI sandbox is **policy-based** — choose a preset level, no custom logic
- Other tools' rules are **advisory** — AI "should" follow but "can" violate

Cline takes a third path: Plan/Act dual-mode implementing "social constraints." Plan mode only analyzes without modifying; Act mode requires human approval for each step. Not technically preventing the agent, but ensuring human confirmation through process.

Three constraint philosophies, each suited to different scenarios. **But the existence of constraints itself is the watershed between a harness being a pile of config files versus being an engineering system.**

Martin Fowler's team analysis points to the same conclusion:

> **Birgitta Boeckeler (ThoughtWorks):** Increasing trust and reliability of an agent requires precisely limiting its solution space. More constraints, paradoxically, means more trustworthy.

Same principle as human organizations. A team without rules isn't free — it's chaos. A team with clear boundaries lets members operate confidently within them.

AI agents are the same. The clearer the constraints you give, the better it performs within those constraints. **A harness doesn't limit AI's power — it makes AI's power trustworthy.**

§16 Capability and Memory

Extending What AI Can Do and Remember

Instructions and constraints govern "what rules the agent knows." The capability layer governs "what it can do." The memory layer governs "what it remembers." These two layers determine the agent's ceiling.

Capability: Don't Load Everything

How much an agent can do depends on how many tools you connect. But more tools isn't always better. The context window is finite — every additional tool description means less space for actual work.

Claude Code's Skills system solves this tension. Skills live in `.claude/skills/`, each a .md file describing what the capability does and when to trigger. **They don't occupy context by default — Claude automatically decides whether to load them based on the current task.**

Writing a .md file defines a new capability. No code needed, no SDK. The lowest-barrier extension method among all AI coding tools.

My own skills directory has dozens: one for social media illustrations, one for Feishu sync, one for video script proofreading, one for information search. Each skill does one thing; naming must let AI instantly understand the intent.

> **核心建议**
>
> Skills design principle: each skill does one and only one thing. The description should read like a one-sentence handoff to a colleague. "Publish the article to Feishu" is better than "Execute the Feishu API document creation workflow." AI decides whether to load based on the description — vague descriptions won't trigger.

MCP: One Protocol Connecting Everything

Skills are local capabilities. What if the agent needs to connect to the outside world?

Model Context Protocol (MCP) is currently the closest thing to a standard. One protocol letting AI coding tools connect to databases, APIs, web pages, GitHub, Jira, Slack. Claude Code, Cline, and Windsurf natively support it. **MCP is becoming the USB port of AI coding tools.**

Block's (formerly Square) open-source Goose coding agent has taken this path furthest. Through MCP, Goose connects to 3,000+ services — from GitHub to Google Drive to Docker to Kubernetes. Stripe's

Minions system also relies heavily on MCP and tool extensions, auto-generating and merging over 1,300 PRs weekly.

Codex CLI's tool definition capability is more limited, focusing on code read/write and command execution. Cursor uses `@docs` to reference external documentation as context. GitHub Copilot has an Extensions ecosystem. Aider is the purest — no extensions, just code editing.

Tool Design Principles

Anthropic introduced an underrated concept in Building Effective Agents: **Agent-Computer Interface (ACI)**. Investing effort in tool documentation and testing matters as much as investing in UI design.

What this means:

推荐

Tool naming lets AI understand intent: `search_knowledge_base`
Each tool has clear parameter descriptions and examples
Tool does one thing and returns results
Error messages tell the agent what went wrong

不推荐

Vague naming: `process_data` (what data? processed into what?)
Parameter descriptions missing, AI guesses
One tool doing five things with a pile of side effects
Errors just say "failed" with no explanation

LangChain's LLMToolSelectorMiddleware makes a clever optimization: using a fast LLM to pre-filter which tools the current task needs, avoiding stuffing all tool descriptions into context. When tools multiply, **selection itself needs to be engineered**.

Memory: The Next Battlefield

As of early 2026, AI coding tools' memory capabilities vary enormously. Most are still at the static file stage.

Tool	Auto Memory	Manual Memory	Cross-Session Persistence
Claude Code	auto-memory (auto-saves observations)	MEMORY.md + CLAUDE.md	Yes
Windsurf	Cascade Memories (auto-generated)	Rules	Yes
Cline	Memory Bank MCP (config required)	.clinerules	Yes
Codex CLI	None	AGENTS.md	Static files
Cursor	None	.cursor/rules/	Static files
GitHub Copilot	None	instructions.md	Static files
Aider	None	CONVENTIONS.md	Static files

Only three tools have dynamic memory; the rest rely on manually maintained instruction files. **Memory systems exist at three levels: AI decides what's worth remembering, explicit management via tool calls, and human manual file updates.** The gap is obvious at a glance.

Claude Code's Memory System

Claude Code has three complementary memory mechanisms.

auto-memory: Claude automatically saves useful observations to `~/.claude/projects/<hash>/memory/`. No need to say "remember this" — it judges what's worth keeping on its own.

MEMORY.md: User-maintained long-term memory. My MEMORY.md records tool preferences, project backgrounds, common mistakes, publishing workflows. I periodically trim it down, keeping it under 100 lines.

CLAUDE.md itself: Project-level memory. Boris Cherny's team uses `@.claude` tags in PRs to update CLAUDE.md. Every new rule is an institutionalized record of a past agent mistake. He calls it compound interest engineering.

Anthropic also released a public beta of a memory tool, letting agents maintain knowledge outside the context window. When Claude played Pokemon, it maintained precise step counts. After context resets, it read its own notes and continued multi-hour sequences seamlessly. This capability is essential for long tasks.

Knowledge Base Management

Memory addresses "what the agent remembers"; a knowledge base addresses "where the agent finds specialized knowledge."

Using my writing project as an example, the knowledge base is organized by topic:

```
_knowledge_base/
├── INDEX.md
├── tech-tools/
├── industry-figures/
├── product-launches/
├── reports/
└── methodology/
```

After each research session, findings are saved to the appropriate category with source URLs and dates. The agent auto-retrieves relevant files when writing articles. **A knowledge base isn't a pile of materials — it's an index the agent can query anytime.**

The OpenAI Codex team was more aggressive: creating self-contained design documents called ExecPlans, migrating Google Docs planning into the code repo, converting Slack decisions to markdown in the repo. The repository must be the single source of truth. **What the agent can't see doesn't exist.**

Context Management: More Isn't Always Better

Both memory and knowledge bases stuff things into context. But context windows have physical limits.

Anthropic discovered a phenomenon called context rot: **the more context tokens, the worse the model becomes at accurately recalling information.** There's a noticeable performance ceiling at about 1 million tokens — beyond that, performance degrades significantly regardless of how large the technical context window is.

There's an even more peculiar finding. Sonnet 4.5 was the first model to become aware of its own context window. It would pre-emptively wrap up when sensing it was approaching the limit, with "very precise but wrong" estimates of remaining tokens. Anthropic calls this context anxiety.

> **Context anxiety was severe enough that compaction alone couldn't sustain long-task performance. Context reset became essential in the Sonnet 4.5 era.** By Opus 4.5, this behavior disappeared on its own. Model progress simplifies harness complexity.

Three strategies for dealing with context limits:

Strategy	Mechanism	Best For
Compaction	Summarize earlier conversation, continue on shortened history	Long tasks where continuity matters
Context Reset	Completely clear window, start new session with structured handoff	When context anxiety is severe
New Session	Start a fresh conversation for task switches	Between unrelated tasks

Claude Code's official documentation gives an intuitive rule: **if you've corrected Claude more than twice and it's still wrong, clearing and starting fresh works better than continued corrections.** Once context is polluted by failed approaches, building on top of it only makes things worse.

Capability and Memory Comparison Overview

A cross-tool comparison for these two layers:

Dimension	Claude Code	Codex CLI	Cursor	Windsurf	GitHub Copilot
Capability Extension	Skills + MCP + Hooks	Desktop App + GPT-5.3	Background Agents + BugBot	MCP	Agent Mode + Extensions
No-Code Extension	Skills (pure .md files)	None	None	None	None
Dynamic Memory	auto-memory	None	None	Cascade Memories	None
Cross-Session	Three complementary systems	Static files	Static files	Workspace-bound	Static files
Context Mgmt	/compact + /clear	Mid-task adjustable	Composer context	M-Query retrieval	Agent mode self-managed

The capability layer trends toward MCP standardization. The memory layer is still fragmented. Both layers will rapidly converge in the next year — agents without memory are simply too painful.

§17 Orchestration: Running Ten Horses at Once

Multi-Agent Coordination

A problem one agent can't solve, ten agents might not solve either. But with the right orchestration, ten agents can accomplish what one agent never could. This chapter covers when to use multi-agent and how.

From One Horse to a Horse Team

All previous chapters discussed a single agent's harness. But in real projects, many tasks exceed what one agent can handle.

Not because it isn't smart enough. Context windows are a physical limit. One agent simultaneously handling frontend, backend, database, testing, and documentation — once the context fills, performance drops off a cliff. Anthropic's data is clear: about 1 million tokens is the ceiling.

So you need multiple agents, **each with its own context window, handling its own specialty**. The question is: who coordinates them?

Boris's 10-15 Concurrent Sessions

The most basic orchestration: humans as the orchestrator.

Boris Cherny (Claude Code creator) routinely maintains 10-15 concurrent Claude Code sessions. Five in terminals numbered 1-5, with shell aliases (za, zb, zc) for quick switching. Five to ten in browsers. Plus mobile sessions started in the morning and checked later.

Each session runs on an independent git worktree — no code conflicts. Some engineers keep a dedicated "analysis" worktree solely for logs and queries, writing no code.

> **This isn't some advanced architecture — it's using a human as the orchestrator.** But Boris says this is his team's single biggest productivity unlock.

The process:

1. **Create multiple worktrees**
   Each task gets an independent working directory, code fully isolated.

2. **Launch a Claude Code session per worktree**

   Terminal numbering for easy switching, OS notifications for critical sessions.

3. **Assign tasks**

   Each session handles an independent module or feature, no interference.

4. **Human coordination**

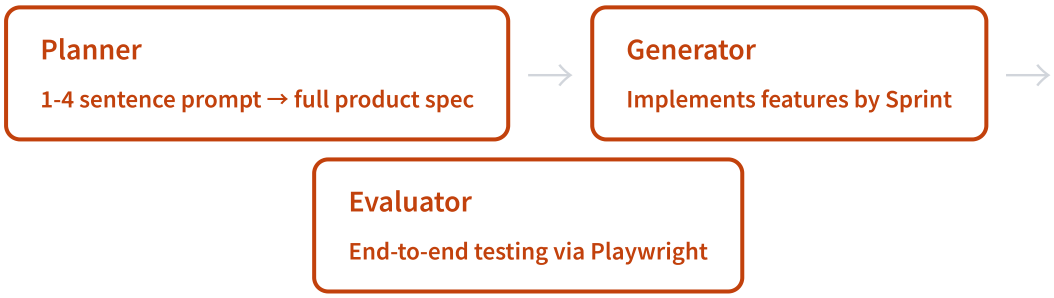
   You switch between sessions, review results, resolve conflicts, merge code.

Upside: fully controllable. Downside: you must keep watching; inter-task dependencies exist only in your head.

Anthropic's Three-Agent Architecture

Don't want to be the orchestrator? Anthropic offers a reference answer.

They built a three-agent system inspired by GANs:



Planner receives short prompts and expands them into complete product specifications. Focus on product context and high-level technical design — not implementation details. Overly detailed technical instructions cascade errors downstream.

Generator implements one feature per Sprint. Before each Sprint begins, Generator and Evaluator negotiate a Sprint Contract — agreeing on the definition of "done." This contract bridges the gap between user stories and testable implementation.

Evaluator is the architecture's soul. Using Playwright MCP, it interacts with the running application like a real QA engineer — testing UI, API endpoints, database state. Scores against preset criteria including design quality, originality, and craftsmanship.

The architecture's core insight bears repeating:

> **Engineering a rigorous independent evaluator is far easier than teaching a generator to self-critique.** Role separation creates adversarial dynamics — a skeptical Evaluator provides critical feedback, helping the Generator break through plateaus.

Anthropic's cost comparison experiment:

Approach	Time	Cost	Result
Single Agent (Solo)	20 min	$9	Core features broken
Three-Agent (Full Harness)	6 hours	$200	Fully functional application

Over 20x more expensive, but the single-agent output was fundamentally unusable. **Not paying more for better — paying because without it, nothing works.**

Interestingly, after upgrading from Sonnet 4.5 to Opus 4.6, the Sprint mechanism was completely removed. The model natively handles long tasks; the Evaluator shifted from per-Sprint to single end-of-run evaluation. **Model progress simplifies orchestration.**

You Can Use Planner/Generator/Evaluator Too

The three-agent architecture looks heavy, but the core idea simplifies to daily use.

1. **Planner: Use Plan Mode**

   For complex tasks, enter Claude Code's Plan Mode (Shift+Tab twice) for detailed implementation planning. Boris suggests: one Claude writes the plan, a second Claude reviews it as a "staff engineer."

2. **Generator: Switch to Normal Mode**

   After plan confirmation, switch to auto-accept mode for coding. When things go wrong, switch back to Plan Mode to re-plan instead of pushing through.

3. **Evaluator: Open a New Session to Review**

   After completion, open a brand new conversation, paste the result, and say: "Find all problems." A fresh-context AI has no bias toward its own code, catching many issues the first AI missed.

This is the poor man's three-agent architecture. No complex system required — three session windows are enough.

Agent Teams: Claude Code's Built-in Multi-Agent

Claude Code's Agent Teams is currently the only solution supporting direct inter-agent communication.

One session serves as team lead, assigning tasks and synthesizing results. Teammates can communicate directly without routing through the lead. Each agent has an independent context window.

Compared to Boris's manual approach, Agent Teams' advantage is **automatic coordination**. Describe a large task, and the lead breaks it down, assigns, and collects results on its own. Suited for parallel research, independent module development, and competing hypothesis debugging.

Cursor 2.0 supports up to 8 parallel AI agents — the most in quantity — but each works independently with **no communication mechanism**. Eight horses each running their own way, with no harness connecting them.

Codex CLI's Subagents inherit the parent agent's sandbox policy and approval settings, running within the main agent's context, only reporting upward. Suited for exploratory subtasks, not for long independent work.

Writer/Reviewer Parallel Pattern

Multi-agent doesn't require complex architecture. The most practical pattern may also be the simplest: one writes, one reviews.

Anthropic officially recommends: one Claude writes code, another Claude with fresh context reviews. Avoids bias toward one's own code.

This pattern scales. For batch processing:

```
for file in $(cat files.txt); do
  claude -p "Migrate $file from React to Vue. Return OK or FAIL." \
    --allowedTools "Edit,Bash(git commit *)"
done
```

Each file processed independently; failures don't affect others. **Parallelism depends on how many processes you're willing to run simultaneously.**

Stripe's Pipeline Orchestration

The enterprise orchestration benchmark is Stripe's Minions system.

Engineer sends a Slack message, walks away, returns to a ready PR. Over 1,300 PRs weekly, all zero human-written code.

Key infrastructure: each Minion runs on standardized AWS EC2 instances (devbox) pre-loaded with Stripe's complete code tree, pre-warmed Bazel and type-checking caches. **Launching a devbox from the warm pool takes under 10 seconds.**

Stripe's core insight: the #1 reason Minions works has almost nothing to do with the AI model — it's infrastructure built for human engineers over many years before LLMs existed. Comprehensive test coverage, synthetic end-to-end tests, blue-green deployment for rapid rollback. **Harness is built on infrastructure; without good infrastructure, harness can't stand.**

Every AI-generated PR still requires human review. But review heavily leverages automated confidence signals. Humans are still in the loop — just in a different position.

When to Use Multi-Agent

Multi-agent orchestration is tempting, but most scenarios work fine with a single agent. Anthropic's guidance from Building Effective Agents:

> **If a single LLM call can solve it, don't use an agent. Agents are for open-ended problems where you can't predict the number of steps or hardcode a fixed path.**

Same logic one level up: if a single agent can handle it, don't use multi-agent.

Scenario	Recommended Mode	Rationale
Fix a bug, add a feature	Single Agent	Context sufficient, no parallelism needed
Refactor a module	Single Agent + Plan Mode	Needs holistic view; splitting loses context
Modify 5 independent modules	Manual multi-session (Boris style)	Independent tasks, no inter-agent communication needed
Write code + review code	Writer/Reviewer dual sessions	Eliminates self-review bias
Complex full-stack app from scratch	Planner/Generator/Evaluator	Task spans front+back end, needs planning and verification
Large-scale migration/batch	Pipeline script orchestration	Highly repetitive, parallelizable
1,000+ PRs/week enterprise scale	Stripe Minions-style platform	Requires dedicated infrastructure

Orchestration complexity should match task complexity. If three session windows can handle it, you don't need Agent Teams. **Start with the simplest approach; upgrade when you actually need to.**

§18 Experience Engineering: Who Will Design the Next Harness

Reflections on the Future of Human-AI Collaboration

This chapter doesn't give answers.

Kief Morris on Martin Fowler's team drew a diagram placing humans in AI coding at three levels.

In the loop: You review agent output line by line, manually fix code you're unhappy with. Human inside the loop, everything passes through your hands.

On the loop: You don't look at the code itself — you build and improve the harness. Specifications, quality checks, workflow guidance. You manage the reins, not the legs.

Out of the loop: You just say what you want, and the agent handles it. Vibe coding.

Morris said something worth pausing on:

> **The difference between "in the loop" and "on the loop" is most visible when you're unhappy with results.** An in-the-loop person goes and fixes the code. An on-the-loop person goes and fixes the harness, so it produces better results next time.

On the loop sounds like a better way to work. You stop repeating labor and instead improve the system itself.

But there's a catch.

If people are moved to on the loop — or even out of the loop — too early, who will design the harness in the future?

Morris's point: if newcomers never touch code details from day one, only operating agents at a high level, then when the harness breaks and someone needs to understand what's happening underneath — who's there?

Martin Fowler put it more sharply in a podcast:

A junior developer's most important attribute isn't what they can produce today — it's that they can grow into a senior developer.

If AI replaces juniors' output but simultaneously strips them of growth paths, that's not efficiency — it's mortgaging the future.

This concern isn't baseless. Shopify expanded their intern program from 25 to 1,000 people, reasoning that interns use AI in more interesting ways. Block laid off 40% of staff citing AI-driven efficiency. Two directions, same problem: the middle layer is disappearing.

Mitchell Hashimoto can write Ghostty's harness well because he understands every detail of terminal emulators. Every line in that config file corresponds to a past agent mistake, and he knows why those mistakes are mistakes.

OpenAI's 3 engineers can drive a million lines of code output because they know what good architecture looks like and what will explode in three months.

Kent Beck can write that TDD-driven CLAUDE.md because he invented extreme programming, spending decades understanding what good code looks like. The first two attempts failed — too much complexity accumulated and AI got completely stuck. Only the third try found the right intervention point.

People who design good harnesses all have deep domain experience. The question is where that experience comes from.

I'm an interesting sample myself.

I've never handwritten code. Every product was AI-written. My app Kitten Light topped the App Store paid charts with over a million users. My harness started from zero, growing bit by bit through AI interaction, with no programming experience to transfer.

But I need to be honest with myself.

I can design harnesses not because I naturally understand system design. It's because in thousands of hours collaborating with AI, I observed its behavior patterns. When it cuts corners, when it hallucinates, when it needs hard constraints rather than gentle reminders.

This judgment doesn't come from coding experience, but from another kind: the experience of repeatedly wrestling with AI.

The question is: can this experience be taught?

I can't say for sure.

I know how to write CLAUDE.md, but if asked to teach someone why I write it this way, I'd get stuck. Many decisions are made by intuition. Intuition comes from mistakes. Mistakes come from massive repetition. Repetition comes from time.

This is really the same as an old programmer saying "write tens of thousands of lines and you'll naturally get it."

So the question may not be whether "coding experience" can be replaced.

Rather: regardless of what kind of experience you accumulate, enough experience itself is the prerequisite for designing harnesses. No shortcuts. Just a different track.

The old track: writing code, debugging, refactoring, getting woken at 3am by production incidents.

The new track: writing CLAUDE.md, configuring hooks, getting driven insane by AI hallucinations at midnight.

The shape of pain changed. The total amount of pain probably didn't.

Perhaps what Martin Fowler should really worry about isn't "nobody writes code anymore."

It's "nobody is willing to spend enough time making enough mistakes."

Boris Cherny said: an engineer's core contribution isn't code — it's judgment. Judgment about what to build, how to verify, when to trust output, when to push back.

Where does judgment come from?

From having done it enough times to know which paths are dead ends.

84% of developers are already using or planning to use AI tools. Anthropic's report says engineers use AI for 60% of their work but can fully delegate only 0-20% of tasks. That middle 40-60% is the human-AI collaboration zone. The soil where judgment grows.

Dario Amodei said in March 2026: **within 6 months, 90% of code will be written by AI.**

If he's right, what happens to that soil?

Karpathy is already living it. Since December 2025, he no longer writes code by hand. His AutoResearch project — 630 lines of code plus a markdown prompt — ran 700 experiments in 2 days. One person plus one harness, doing the work of a team's several weeks.

But Karpathy can do this because he's Karpathy. He knows which experiments are worth running, which results are worth pursuing, which findings are noise. **That markdown prompt works because the person who wrote it has decades of research intuition.**

What if that soil gets fully automated away?

I don't know the answer.

I only know my own harness grew over thousands of hours of wrestling. If someone asked me how to quickly learn harness design, the first thing I'd say is: you can't do it quickly.

By the way, one more thing. Starting April 24, 2026, GitHub Copilot's free and Pro users will have their interaction data used for model training by default. How you collaborate with AI, your harness approach, your instruction file style — all become training data for the next generation of models.

You're teaching AI how to work. AI is learning how you teach.

In this loop, who designs the harness may matter more than who rides the horse.

I'm not sure about that either. I'll leave it for you to think about.

Harness Engineering

The Complete Guide to AI Agent Harness Design

Huashu

Engineering practices for the age of AI coding: from origins to framework, 7 real case studies to hands-on guide

Harness AI with reins — keep every horse running in the right direction

X/Twitter: @AlchainHust · YouTube: @Alchain · Bilibili · huasheng.ai

Created by Huashu · v1.0 · 2026