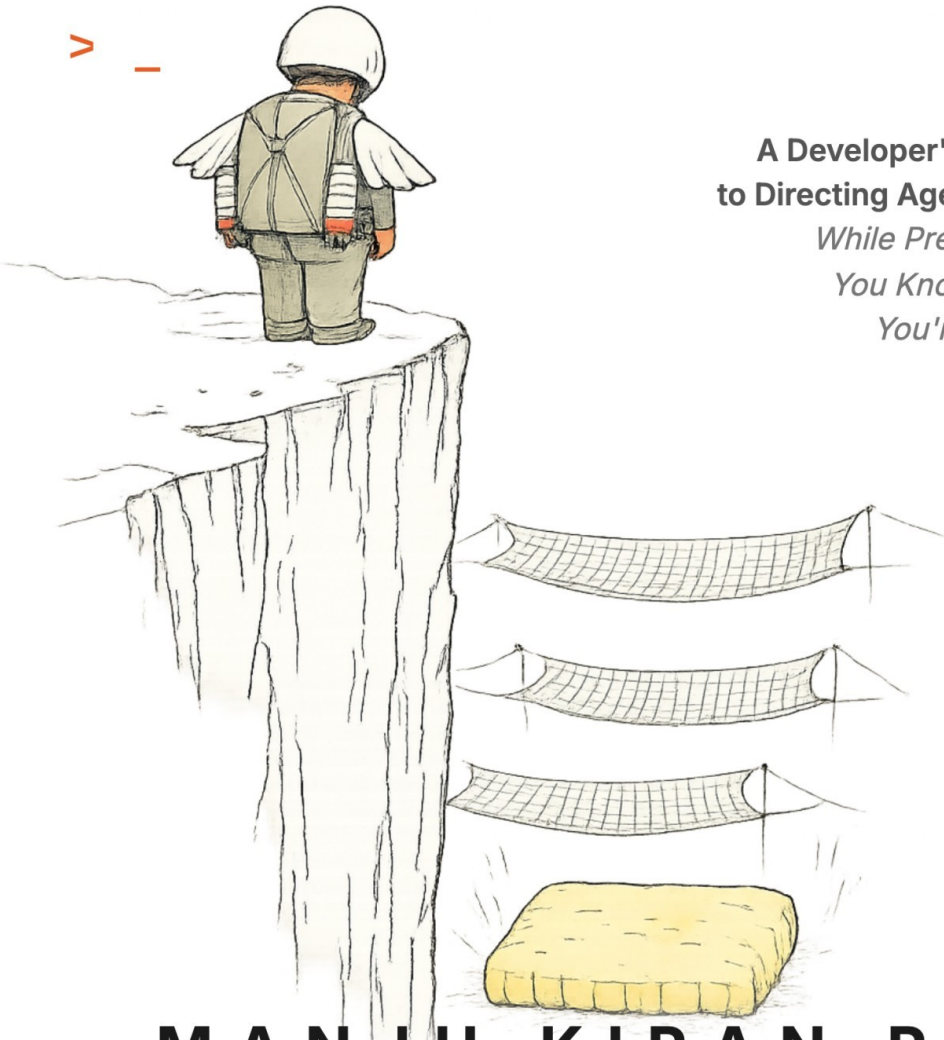
IT AI'nt THAT HARD

> _

A Developer's Guide to Directing Agentic AI,
While Pretending You Know What You're Doing



MANJU KIRAN R

It AI'nt That Hard

A Developer's Guide to Directing AI, While Pretending You Know What You're Doing

Manju Kiran R

2026

Contents

Copyright

ISBN: 978-93-5717-532-6

For permissions requests, contact: contact@manjukiran.com

Trademark Acknowledgements

Claude, Claude Code, Claude Opus, Claude Sonnet, and Claude Haiku are trademarks of Anthropic, PBC. GitHub and GitHub Copilot are trademarks of GitHub, Inc. Linear is a trademark of Linear Orbit, Inc. All other

product names, logos, and brands mentioned in this book are property of their respective owners.

This book is not affiliated with, endorsed by, or sponsored by Anthropic, PBC or any other company mentioned herein. All opinions expressed are the author's own, based on personal experience.

Financial Disclaimer

This book describes software development practices and processes. Nothing in this book constitutes financial, investment, or trading advice. The software application described is the author's personal project, used for illustrative purposes only. Any trading-related examples are included solely to demonstrate software engineering concepts and should not be interpreted as recommendations or strategies.

Currency and Pricing

Product features, pricing, and model names described in this book were accurate as of early 2026. Technology products evolve rapidly; refer to the respective vendors' websites for current information.

AI Assistance Disclosure

AI tools were used to review and critique drafts of this manuscript during the editing process.

Author's Note

The technical content in this book - the tools, the workflows, the costs, the code, the principles - is drawn

from my actual experience building software with AI assistance. The failures are real. The costs are real. The protocols exist and I use them daily.

The narrative surrounding that content is creative non-fiction. Conversations have been reconstructed from memory and, in some cases, composed to carry arguments that emerged over weeks of scattered kitchen-table discussions into scenes that work on the page. My wife is a real developer with her own relationship to these tools, but I have taken liberties with specific dialogue, scene-setting, and the occasional anecdote to serve the narrative. Some events have been compressed, reordered, or dramatised. Some domestic details are invented because the real ones were less interesting or I simply can't remember whether it was a Tuesday or a Thursday.

Judge the principles on their merits. The stories are true in the way that matters - the lessons they carry - even where the specific words spoken are my best reconstruction rather than gospel.

A Note on Brand Names

Throughout this book, I use "the AI," "the agent," and "my AI assistant" interchangeably with specific product names. Where I name a specific product, it's because the story requires it - a product-specific feature, a pricing discussion, or a particular technical behaviour. Where I use generic terms, the lessons apply regardless of which AI coding tool you use. The principles in this book are tool-agnostic, even if my experience wasn't.

First Edition

CHAPTER 3

The Dream That Died at 3 AM

The Vision

I had a dream.

Not the Martin Luther King kind. The dangerous kind - the kind that makes you text yourself at midnight so you don't forget it.

Picture this: It's 6 AM. I'm asleep. The toddler is asleep. The house is quiet in that fragile way that means nobody has woken up demanding dinosaur videos yet. And somewhere in the corner of my office, my machine is not asleep. My machine is working.

A daemon I've written polls GitHub every thirty seconds. A new PR appears. The daemon springs to action, spawns a Claude session, and Claude reviews the pull request - checks the code, validates against acceptance criteria, runs tests, leaves detailed comments. By the time I check my phone over breakfast, there's a notification waiting: "PR #127 reviewed. 2 comments. Ready for merge."

All while I dreamed of electric sheep.

But wait. The dream was bigger than that.

What if the PR had issues? In my grand vision, the review bot wouldn't just comment. It would talk to the implementing agent. A conversation between AI systems, happening in git-ignored threads while I remained blissfully unconscious. "Hey, you missed edge case X." The implementing agent fixes, pushes, requests re-review. The reviewer checks again. Review-fix-review-approve. AI agents resolving their differences like mature adults while I slept.

Automated. Continuous. Self-healing.

I spent four days building this system. Then I discovered it was architecturally impossible.

Let me walk you through my reasoning, because honestly, it was sound. That's the frustrating part.

The Architecture That Made Perfect Sense

The logic was impeccable. The execution was professional. The entire foundation was rotten.

The architecture had three components. First, a PR monitor written in Node.js - clean code, environment variables, proper logging. I even added graceful shutdown handlers because my first software mentor once told me that gracefulness matters, in both developers and their products. I remembered that lesson. I forgot some other, more important ones.

```
const config = {
  repos: [
    'myorg/backend-api',
    'myorg/data-services',
    'myorg/frontend-app'
  ],
  pollInterval: 30000,
  claudePath: '/usr/local/bin/claude'
};
```

It looked professional. It felt professional. Like a real system built by a real engineer who knew what he was doing.

Second component: a trigger script that would invoke Claude with a command. Just a shell script. Nothing fancy.

```
claude - command "/pr-review ${PR_URL}"
```

Third component: my /pr-review command, which was already mature at this point. Fetch PR details, create an isolated workspace, run local git diffs, validate against acceptance criteria, leave review comments. All the pieces existed. I just needed to connect them.

Like Lego. What could go wrong?

GitHub webhook or polling detects PR. Spawn Claude Code session. Execute the review command. Post results. Loop. Simple. Clean.

Wrong.

Day one was foundations. I built the Node.js service with the care of someone constructing a cathedral. The monitoring worked. The polling worked.

Everything worked, right up until the part where it needed to actually talk to Claude.

Day two was when I should have stopped.

The Integration That Wouldn't

My first attempt at Claude integration was direct CLI invocation. Just call Claude with a command, like any other tool. Problem: Claude starts an interactive session. It expects human input. It sits there, waiting,

cursor blinking, while my daemon stares at it and nothing happens.

Second attempt: environment variables. Maybe Claude would read a command from an env var? Problem: that's not how Claude works. I was guessing at interfaces that didn't exist.

Third attempt: heredoc input. Pipe the command into Claude's standard input.

```
claude << EOF
/pr-review https://github.com/myorg/backend-
api/pull/42
EOF
```

Problem: Claude's session model doesn't accept piped input to commands. The characters went in. Nothing came out. Like shouting into a pillow.

My wife was on the sofa that evening with her own laptop - she'd been debugging a CSS specificity nightmare on one of her projects all afternoon. She'd glanced over a few times as I cycled through terminal windows, eight of them, maybe nine, each representing a different failed approach.

"What are you actually trying to get it to do?" she asked, not looking up from her screen.

"Automated PR reviews. The monitor detects a new PR, spawns a Claude session, runs the review command, posts comments. All while I sleep."

"And the spawning bit is the problem."

"The invocation, yeah. I can't find the right way to pass it a command non-interactively."

She closed her laptop lid halfway and studied my terminal windows. "Have you actually tried piping a command to it? Just the basic thing. One terminal, one pipe."

"I'm building the orchestration layer first - the polling, the triggers, the error handling. Once that's solid I'll wire up the Claude integration."

"Manju." She pulled up a terminal of her own. "Last month I spent two days building a caching layer for an API that turned out to have a rate limit of ten requests per hour. Beautiful Redis setup. Completely pointless, because I never checked the constraint that mattered."

"This is different. The architecture is sound."

"The architecture is the top four floors of a building. I'm asking about the foundation."

"I will. After I finish the trigger script."

She looked at me for a long moment - not annoyed, but with the weariness of someone watching a mistake they recognise from the inside. She'd made this class of error before. She could see the shape of it.

"Right," she said. "Crack on, then."

She went back to her CSS. I went back to my trigger script. We both knew how this was going to end.

The Desperation Phase

Maybe I was approaching this wrong. What if I could run Claude in a persistent session and send commands through IPC? What if there was some headless mode I

hadn't discovered, lurking in the documentation like an Easter egg?

I tried screen sessions:

```
screen -dmS claude_review "claude"
screen -S claude_review -X stuff "/pr-review
${URL}^M"
```

I tried expect scripts:

```
spawn claude
expect ">"
send "/pr-review $url\r"
```

I tried things I'm not proud of. Things that belong next to that twelve-word prompt from Chapter 2 and the regex I still don't understand from 2014.

Nothing worked.

At some point around midnight, I asked Claude itself how to automate Claude. The irony was not lost on me. It gave me thoughtful suggestions that didn't work, which felt appropriate somehow. Even the AI couldn't figure out how to automate itself.

The coffee had stopped helping hours ago but I kept drinking it anyway, because admitting the coffee wasn't helping meant admitting I should go to bed, and admitting I should go to bed meant admitting this wasn't going to work, and I wasn't ready to admit that yet.

By 3 AM, I had no choice.

Acceptance

Day four. 3 AM.

I was surrounded by terminal windows. Empty coffee cups formed a small city on my desk. The specific delirium that comes from debugging something you're slowly realising is impossible had settled over me like a fog.

I accepted the truth.

Claude Code is designed for interactive sessions with humans. Not for programmatic invocation. Not for piped commands. Not for daemons that spawn it at 6 AM while developers sleep.

The dream was dead.

I sat there for a while, staring at the code I'd written. Good code. Clean code. Useless code. Four days of careful engineering that would never run in production because the fundamental assumption - that Claude could accept commands from a script - was wrong.

It's a particular kind of defeat, building something that works perfectly except for the one thing it needs to do. Like constructing a beautiful car and then discovering you forgot to include an engine.

Fifteen minutes. That's how long it would have taken to discover this didn't work. Open a terminal. Try to pipe a command to Claude. Watch it fail. Move on with my life. I knew this. On some level, I knew this the whole time.

Instead: four days. Because the architecture was seductive. Three clean components, a mermaid diagram that looked like it belonged in a systems design

textbook, graceful shutdown handlers. When the design is beautiful enough, you stop questioning whether it stands on anything.

My wife had asked the right question on Day 2 - "Have you actually tried piping a command to it?" - and I'd brushed past it because I was in love with the architecture. Every experienced developer has done this. Built the cathedral before testing the soil. I'd spent fifteen years warning clients about exactly this pattern, and I'd walked straight into it because the vision was too pretty to question.

You can't pipe into a conversation. Conversations require conversants. It's like trying to automate a Socratic dialogue. The whole point is the back-and-forth. Take away the back-and-forth and you don't have a more efficient dialogue - you have nothing.

Maybe by the time you read this, my dream is possible. Check your Claude version. If headless mode is there, pour one out for past me.

I told her over breakfast. She was feeding the toddler, who was distributing porridge across surfaces in ways that suggested a future in abstract expressionism.

"The pipe doesn't work," I said. "Interactive sessions only. The whole thing is dead."

She didn't say I told you so. What she said was worse, in a way, because it was generous.

"The review command itself - that works, though? The bit where you actually run the review?"

"Yeah. That part's solid."

"So you built a good review tool and a dead automation layer. Keep the good bit. Bin the dead bit." She wiped porridge off the table. "You've got a command that does a ninety-second review. That's not nothing, Manju. That's actually useful."

The kindness stung more than smugness would have. We'd both made the same class of mistake before - build first, validate later. She'd learned to check the foundations earlier than I had on this particular project. Next time, I'd check first. The expensive lesson was, at least, a permanent one.

Every workflow I built after that night - dev-workflow, child-ticket-workflow, pr-merge, all of them - started with a question that now lives in my bones: does the basic thing work? Not the elegant thing. Not the automated thing. The basic thing. The fifteen-minute test that either lets you build with confidence or saves you four days of cathedral construction on sand.

The dream of automated reviews while sleeping died at 3 AM on a Thursday. What took its place was a manual-but-fast review command. One line in the terminal:

```
/pr-review https://github.com/myorg/backend-
api/pull/42
```

Ninety seconds later, a thorough review. Not automatic. But fast enough that "manual" doesn't hurt. The first time I ran it and watched a review appear before my toast popped, I felt something I hadn't expected - not the triumph of the dream, but the quieter satisfaction of a thing that actually works.

I saved the dead code to a folder called pr-review-bots - deprecated, because I couldn't quite bring myself to delete it. Architects rarely can.

Summary

A senior iOS developer watches a four-minute demo, feels fifteen years of craft evaporate, and does what any rational person would do at midnight with a sleeping toddler upstairs: panics.

It AI'nt That Hard is the story of what happened next. Not the polished version. The real one - kitchen-table arguments about token budgets, forty-seven merge conflicts on a Monday morning, an agent that quietly passed empty tests for three days while nobody noticed, and the slow reckoning with a job that changed shape while I was still learning to hold it.

My wife - a developer herself, working from the other end of the same kitchen table - became the book's unofficial control group. Where I built elaborate multi-agent orchestration systems and 262-line workflow protocols, she used Copilot with a light touch, switched to the expensive model only when stuck, and shipped perfectly good client work without the existential crisis. Her approach wasn't wrong. Neither was mine. The gap between them is where most of this book lives.

This is a memoir about infrastructure, failure, and the uncomfortable question at the centre of it all: when the machines do the building, what's left for the builder?

The failures are real. The PR that passed every check but was missing two critical files. The agent that nearly deleted production config as a helpful cleanup. The three-day silent failure that smiled and waved. The defensive code patterns I kept writing because AI defaults to tolerating mismatches instead of fixing them.

The solutions are real too. Workflows that enforce discipline when a sleep-deprived parent at 2 PM would

shortcut. Testing agents more paranoid than any human QA. Living documentation that updates itself because nobody else ever will. Contracts written before code, because agents who can't hear each other will invent three different message formats for the same feature.

The questions don't have tidy answers. What happens to craft when the craft is automated? Where does judgement come from if juniors never get to build? Is directing AI systems still programming, or is it management with better tooling?

For developers navigating the shift. For the honestly curious. For anyone who suspects there's a space between "AI will replace us all" and "AI is just autocomplete" where the interesting work actually happens.

There is. It's just messier than the demos suggest.

The Chapter Overview

Part I: The Wake-Up Call

Chapter 1: Fifteen Years to Fifteen Minutes - I watch an AI build in four minutes what took me a day, and have a midnight crisis about it with my wife.

Chapter 2: My First Prompt Was Embarrassingly Bad - Twelve words, no context, terrible output. The oldest lesson in computing, relearned the hard way.

Chapter 3: The Dream That Died at 3 AM - I spend four days building an autonomous PR review system. It was architecturally impossible.

Part II: Building the Infrastructure

Chapter 4: Setting Up the Orchestra Pit - CLAUDE.md files, context cascading, and the invisible plumbing that separates a demo from a system that survives production.

Chapter 5: From Paragraph to Protocol - A fifteen-line workflow instruction becomes 262 lines across nine phases. Every line earned through disaster.

Part III: The Hard Problems

Chapter 6: When Agents Collide - Two agents, one directory, forty-seven merge conflicts, and my wife diagnosing the carnage over coffee.

Chapter 7: The Ticket That Broke Everything - Seventeen acceptance criteria. The agent implemented two. I'd already marked it ready for review.

Chapter 8: The Testing Specialist - Building a paranoid QA agent that writes chaos engineering tests I'd have called overkill, until Redis went down.

Chapter 9: The Economics of Amnesia - Seventy percent of my tokens wasted on the AI re-reading the codebase every session. My wife does the maths over dinner.

Chapter 10: Living Documentation - I build an endpoint that already exists because the docs were three weeks stale. Documentation that lies is worse than no documentation at all.

Part IV: Coordination

Chapter 11: The Child Ticket Dance - Three agents build three incompatible implementations of the same feature. The fix isn't technical - it's contracts before code.

Chapter 12: The GPS in the Tesco Car Park - The agent finds a mismatch between contract and implementation - and instead of fixing the source, teaches the consumer to tolerate the error. Why defensive code is worse than no fix.

Chapter 13: PR Review in Ninety Seconds - The day I nearly approved a PR missing two critical files. Automated review that catches what humans miss: absence.

Chapter 14: The Process That Runs Itself - Sprint planning became something that happens in tickets, not meetings. I'm not sure if that's progress or loss.

Part V: Oversight

Chapter 15: Watching the Watchers - For three days, the backtest agent ran empty tests and reported success. Observability for systems that fail quietly, not cleanly.

Chapter 16: The Hard Limits - An agent nearly deletes production config as a helpful cleanup. Guardrails built incident by incident, because capability without constraint is chaos.

Part VI: Reckoning

Chapter 17: The Oversight Economy - My wife asks: "What do you actually do, then?" I don't have a clean answer. Neither does she.

Key Concepts

The Oversight Economy: The new model where the scarce resource isn't human labour - it's human judgement. The job isn't "write code." The job is "ensure the right code gets written."

Contract-First Development: Explicit agreements between systems before anyone writes a line of code. Essential when agents can't overhear each other in Slack. Three agents, three message formats, one expensive lesson.

Living Documentation: Documentation that updates automatically as part of the development workflow. Because documentation that isn't current isn't documentation - it's a lie that looks like the truth.

Context Cascading: Layered configuration from global to directory-specific, like CSS specificity for AI context. Broad to narrow, general to specific, each layer scoped to its purpose.

The Control Group: My wife's lighter approach to AI tooling - pragmatic, unsentimental, effective - as counterpoint to my elaborate infrastructure. A reminder that the right amount of process is less than you think.

Trace, Don't Patch: When systems disagree, fix the source, not the consumer. Never write defensive code that tolerates wrong data. The contract is truth. One name, one type, one scale.

Guardrails Over Guidelines: Hard limits enforced by code, not soft guidelines enforced by hope. Forbidden commands, approval gates, rate limits, scope boundaries - built incident by incident because each one has a story.

Who Should Read This

- **Senior developers** adopting AI tools and wanting to skip the failures I didn't
- **Team leads** trying to establish AI-assisted workflows that survive contact with reality
- **Solo developers** building systems too complex for one person without AI leverage

- **Developers' partners** who want to understand why someone is muttering at a laptop at 2 AM
- **Anyone sceptical** of AI hype who wants an honest account of what works, what doesn't, and what nobody talks about

Who Shouldn't Read This

- Readers wanting a tutorial on specific AI tools (the interfaces will change before the ink dries)
- Readers wanting confirmation that AI will replace developers (it won't, but the job will change shape)
- Readers wanting confirmation that AI is useless hype (it isn't, but the demos oversell it)
- Anyone allergic to British spelling, dry humour, kitchen-table domesticity, or honest admission of failure

"The book for every senior developer having a 2 AM panic attack about AI"

You've spent fifteen years building expertise. You've debugged the impossible bugs. You've survived the incidents that became cautionary tales.

And now you're watching AI write code faster than you can type.

It AI'nt That Hard is the story of one developer's journey from midnight panic to genuine mastery — and a practical system for every senior developer facing the same crossroads. Born from six months of real experimentation, real failures, and real production code.

INSIDE, YOU'LL DISCOVER

- Why your experience becomes leverage, not liability
- The 262-line protocol born from forty-seven merge conflicts
- How to run an AI software agency from your laptop
- The economics of AI development nobody talks about
- Why the oversight economy needs you more than you think

Manju Kiran Ravishankar is a software developer with fifteen years of experience building mobile applications for banking, healthcare, and trading platforms. Originally a civil engineer who wandered into tech, he spent 2024 doing what most senior developers were afraid to: actually testing whether AI would replace him. It didn't — but it did replace his job description. He lives in the UK with his wife Rekha, also a developer, and their toddler, who remains unimpressed by all of it.

9 789357 175326

ISBN 978-93-5717-532-6

Software Development / Memoir

manjukiran.com

$24.99 US / £19.99 UK