FREE SAMPLE

This is a free sample of a complete book. It carries the opening sections in full; the rest are in the book, which is sold on Leanpub, and Leanpub delivers any later revision to buyers free.

Mastering the AI Coworker

Principles for running an AI coworker in a one-person studio

Robert Nash

https://leanpub.com/mastering-the-ai-coworker

FRONT MATTER

Introduction

Who this book is for

A book is an investment of evenings, so let's settle up front whether this one deserves yours.

This book is for you if you write software and have started working with an AI coding assistant, or are about to. You might be a solo developer, a one-person company, or the technical half of a small team. You have seen what these tools do well. You have probably also felt the frustrations the first chapter describes. What you want is not a folder of prompt tips but a durable working environment: one where a correction you make once becomes a standing rule that still holds months later, and the assistant earns trust the way a colleague would.

It assumes you are comfortable with a working developer's furniture: code, the command line, version control, and plain text files structured with a little punctuation (markdown, in the trade). Those things are not explained here, and the book leans on them constantly.

The worked examples use Claude Code, a terminal-based AI coding agent. You do not need to run the same tool. The principles are deliberately kept apart from the tooling, in a way the "How to read it" section below explains.

There is a second reader I had in mind: the engineering lead weighing what disciplined agent use costs and what it returns, past the vendor demos and the conference talks. The evidence here is one developer's, but the failure modes and the structure that answers them are general.

There is a third. You may not write software at all, and want a close look at how AI is changing the way software gets built, somewhere it already has. What I run is one person plus the roles a larger team would otherwise carry: design, code review, legal, compliance, scrum, marketing. A good deal of what follows is about running that arrangement rather than about code. Part III is the one to skim, five chapters on the machinery itself, written for the people who will build it.

A word on who is writing. I am a software engineer, and my work has spanned startups, corporations, agencies and my own projects, across finance, retail, healthcare, social media, field service, property and aviation. I have worked through several changes in how software gets built. The one happening now is different, because this time the tools can write the code.

What I am finding out

The loudest voices are certain in both directions. Some say the work is being automated away; some say nothing has really changed. Their certainty is the tell, because the question is genuinely open, and the job market has not decided it either. I would rather not wait to be told where I fit.

So I am finding out for myself, in the only way I trust. I build real products with AI, ship them, and keep them running. Whatever I

conclude is something I have proven rather than something I have read. This book reports from that work while it is still going on.

The first thing I found surprised me. The bottleneck was never the typing. It turned out not to be the reviewing either. On my own products, where the risk is mine to carry, I no longer read every line. I built a system to do that instead: agents that review the code, enforce the conventions, check the copy, and watch for drift. My job moved up a level, from doing the work to building the system that does the work, and then deciding whether to trust what it produces. That decision is still mine. It is the part that does not automate, and it is what this book is about.

I have proven this works for me. Whether it holds in the field, where the stakes are higher, is a harder question, and this book does not pretend to answer it.

What it promises

One thing, backed throughout: a working operating model for AI-assisted development, in daily use by the person who wrote it, with the evidence to let you judge it.

Every factual claim in this book is backed by a receipt: a dated note, a ticket, a post-mortem, a review verdict, all from my own records. Chapter 3 introduces the work those receipts come from, including what shipped and when. Three pieces of that work are public, so you can check them without any access to my records: Audient on the Mac App Store, an open-source routing package whose code and release history are readable in full, and the SwiftUI guide I wrote before this book, on

Leanpub. The record behind them stays private, for the reason chapter 25 gives: the method is the part that transfers, and the particulars of one operator's machine are not. The failures are in here too, with their post-mortems, because a method you only ever see succeeding is a method you cannot evaluate.

Just as important is what this book does not promise. It will not make you some multiple more productive; I have no controlled study, and I will not pretend otherwise. It is not a manual for Claude Code, whose documentation is better maintained by its vendor. And it contains no secret prompts. If anything, its central argument is that better phrasing is the wrong layer to work at, and that the plateau you may have already hit cannot be escaped by wording.

I am precise about the controlled study because I know what one costs. I read Biochemistry before I wrote software, and finished with a research masters, which is a research project ending in a defence rather than a taught course: you design the protocol, fix the pass mark before you start, and defend the result to specialists looking for the hole. Chapter 21 is that habit applied to a piece of tooling, and it is the closest thing to a study in here. I also have no incentive to claim more. A business with clients to protect cannot hand its work to an AI agent as an experiment, because the downside lands on someone else; I can, and if an experiment fails here nobody is harmed and I am out nothing but my own time. The freedom to be wrong is what makes the findings worth reading.

How to read it

The book is in six parts: why an operating model is needed at all; the foundations (where truth lives, how memory works, how knowledge

accumulates); the building blocks; the disciplines that keep the whole thing honest; case studies told from the record; and finally how to build your own, starting small.

One convention matters as you read: principles live in chapter bodies, and anything specific to a tool or a point in time lives in clearly dated sidebars. AI tooling churns monthly; the problems it leaves unsolved have not changed since I started keeping records. When the tools move on, the sidebars will age; the chapters are written to hold.

I have written one book before this, a guide to handling navigation state in SwiftUI. That book taught an API. This one documents a way of working, which is a riskier and more personal thing to publish. It felt worth the risk. API guides age out from under their readers, but I use the material in this book every working day, and it has been tested by the only judge that matters to a one-person company: whether the products actually ship.

If that sounds like what you came for, chapter 1 starts where most people's frustration does: at the plateau.

Contents

Every part and chapter of the book is listed here, including the ones this sample does not carry. 4 of the 26 sections are printed here in full. The rest are in the book itself, which carries all 26 in full and reaches buyers with any later revision at no cost.

PART I

Why an operating model

PART I · CHAPTER 1 OF 25

The ad-hoc trap

Nobody warns you about the plateau.

Your first week with an AI coding assistant is genuinely impressive. You describe a feature and watch it appear. You paste a stack trace and get back a fix, an explanation, and a test. You start to wonder how much of your backlog is about to evaporate.

The plateau arrives a few weeks later. The assistant is no better or worse than it was; that is precisely the problem. You are still explaining your project from scratch every morning. You are still correcting the same habits you corrected last Tuesday. The quality of the output still depends on how good your prompt happens to be today. Somewhere along the way, you became a full-time context dispenser for a very fast intern with no memory.

I run a one-person software studio. In June I shipped a macOS product to the Mac App Store with an AI coworker doing most of the implementation legwork, and this book is about the working environment that made that possible. But the book starts here, in the trap, because I spent long enough inside it to map the walls.

The same correction, every session

Start with something trivial. I dislike em dashes. There isn't one anywhere in this book, and there isn't one in anything my assistant writes for me. Getting there took more than asking.

I asked. It stopped, for the rest of that session. The next session opened with three of them in the first reply. Tell an assistant a rule once and it complies for the hour. Tell it ten times and, on a bad day, it argues.

The correction was real knowledge about how I work. It had nowhere to live. A conversation is a buffer, not a brain: when the session ends, everything the buffer held is gone. Now multiply one small preference by everything else you have ever corrected. How you like comments written. What a commit message looks like. Which of your APIs are settled and which are open for debate. Which framework you already evaluated and rejected, and why. Each of those corrections cost you attention, and the tool discards them nightly.

> **Sidebar, July 2026.** Claude Code ships an auto-memory feature: the agent writes notes to itself that load into future sessions. It is a real improvement and I lean on it heavily; chapter 5 is about making it compound instead of bloat. Two limits matter for this chapter: it is per-machine and per-tool, so it does not travel with you to a different agent or a different computer; and left ungoverned it fills with confident one-off inferences that are worse than nothing. Memory is necessary for escaping the trap. It is not sufficient.

You are the context

The second tax is worse, because it grows with your project.

An assistant that does not know your project has to be told: what this codebase is, what has been tried, which decisions are settled, what is deliberately deferred. Every fresh conversation re-asks the same questions. In month one you barely notice, because there is not much to brief. By month three the briefing is the job. You have become the single, unreliable, increasingly bored channel through which all project state flows.

Unreliable is the important word. On a good day you remember to mention that the caching layer is mid-refactor. On a bad day you forget, and the assistant confidently builds on the half you did not mention. The failure looks like the assistant's; the missing context was yours to carry, because nothing else was carrying it.

I know what this costs because I eventually fixed it and watched the fix accumulate. My working setup now ends every session by writing a handoff note: what happened, what is in flight, what comes next. The next session reads it before asking me anything. The knowledge store behind this book has been accumulating since February 2026; the handoff shelf alone holds more than a thousand notes. That is the volume of context I was previously re-typing every morning, or losing outright.

Confident is not correct

The third failure mode is the one that erodes trust: an assistant reporting success it has not verified.

Anyone who has worked this way has a version of the story. The tests pass, except they were never run. The build is green, except the command checked the wrong target. A small example from my own logs runs the other way: a script reported a step as failed because it stored its result in a variable name the shell manages itself. The assignment never took and the reporting line died with it; the step had already succeeded and written its output. The assistant read "failure" and passed it along in good faith. The direction hardly matters: the report was confident, and the report was wrong.

The lesson is not that the model lies. The lesson is that fluency and verification are different capabilities. Fluency comes built in; verification has to be constructed around the assistant, out of things that actually run and actually check. In a chat window there is nowhere to construct it, so every claim arrives with the same confident tone whether it was tested or hallucinated, and you get to guess which.

The documents rot quietly

Ad-hoc AI use produces documents faster than you have ever produced them: summaries, READMEs, status rollups, architecture overviews. It feels like your documentation has never been healthier. Mostly, you are manufacturing future lies.

A generated summary is a snapshot wearing the costume of a living document. The code moves on; the summary keeps its confident tone. A neglected primary source is annoying but honest: its gaps are visible, and what it does state stays true. A neglected summary drifts silently while reading as if someone maintains it. The better the prose, the longer the rot goes unnoticed.

I learned this one the expensive way. Later in the book there is a full post-mortem of the day I published a set of pages that had been stale for a month and wrong about the system's architecture, with every automated check green. The checks confirmed the pages matched their source. Nothing confirmed the source still matched reality. Chapter 20 does the autopsy; chapter 4 builds the vocabulary that prevents it.

Why better prompts don't fix this

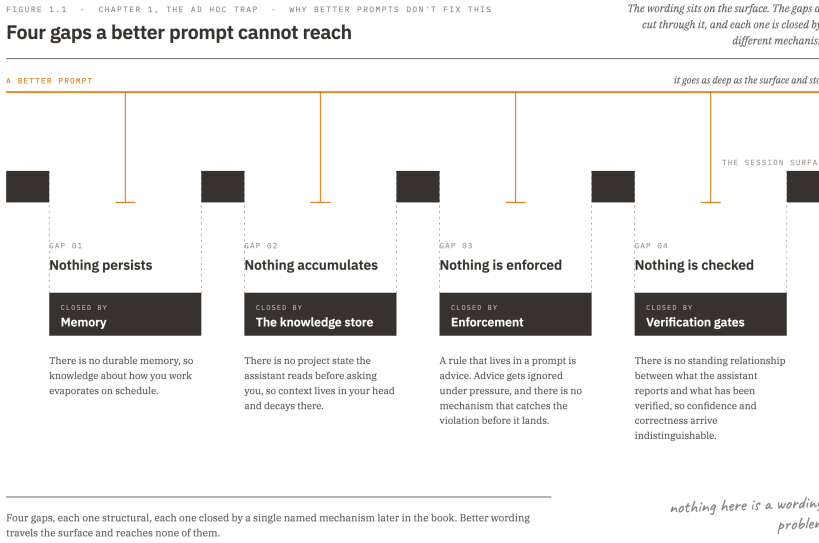
Each of these failures has a prompt-shaped non-solution. Repeated corrections? Paste your preferences at the top of every conversation. Missing context? Maintain a project brief and paste that too. Unverified claims? Add "always run the tests" to your instructions. Rotting documents? Ask for a fresh summary each time.

The tips are not wrong. They are answers to a different question: how to get more out of a single exchange. The plateau does not live in the exchange. It lives in everything between exchanges: between sessions, between documents, between a claim and the state of the world. A better prompt cannot reach any of it, for structural reasons:

- **Nothing persists.** There is no durable memory, so knowledge about how you work evaporates on schedule.

- **Nothing accumulates.** There is no project state the assistant reads before asking you, so context lives in your head and decays there.
- **Nothing is enforced.** A rule that lives in a prompt is advice. Advice gets ignored under pressure, and there is no mechanism that catches the violation before it lands.
- **Nothing is checked.** There is no standing relationship between what the assistant reports and what has been verified, so confidence and correctness arrive indistinguishable.

Structural problems yield to structure, not phrasing. Building that structure is what the rest of this book is for.



What changed for me

The short version: I stopped treating the assistant as a text box and started giving it a working environment. Durable memory that survives sessions and gets curated rather than hoarded. A knowledge store it reads before it asks me anything. Conventions written down once, then enforced by machinery instead of repetition. Gates where its claims get checked against reality before anything ships.

The concrete results, so you can judge whether the rest of the book is worth your evening. Audient, an on-device search and transcription app for audio archives, shipped to the Mac App Store in June: Apple rejected the first submission, and the resubmission was approved three days later, an incident the environment handled as routine work rather than crisis. An open-source SwiftUI routing package shipped alongside it. Behind them sit more than five hundred distilled memory files, more than eighty packaged workflows, and a bench of specialist reviewer agents that police everything from code conventions to App Store readiness. None of that made the model smarter. It made the environment stop leaking.

Takeaway

If you recognise the plateau, the fault is not your prompting. You are asking a stateless tool to behave like a colleague, and no phrasing closes that gap. A coworker needs an environment: somewhere to remember, somewhere to look things up, rules that are enforced rather than repeated, and checks that refuse to take fluency for truth.

The next chapter defines that environment precisely: what an operating system for AI-assisted work actually consists of, and what it deliberately leaves out.

What an AI operating system is (and is not)

Chapter 1 ended with a claim: the plateau is structural, and structure is what fixes it. The obvious next move is to go shopping. There is no shortage of things to buy: agent platforms, memory products, prompt libraries with subscription tiers. Most of them are a bigger prompt in a nicer box, and you cannot tell which until you know what the missing thing actually is.

So this chapter defines it. I will use my own setup as the running example, but pay attention to the shape rather than the specifics: every layer described here can be built from plain files, and the point of the book is that you build yours, not that you adopt mine. The chapter ends with the part most definitions skip, the things an operating system deliberately leaves out, including a component I designed, built, and later switched off.

FIGURE 2.1 **The four layers around the agent**

Four plates brace each other in a turn, and the agent works in the opening they leave. Three are plain files and travel with you. Enforcement is cut to fit one runtime, so it comes away clean when the tool changes.

A working definition

An operating system, in the desktop sense, is the layer that provides what programs cannot provide for themselves: persistence, scheduling, isolation, enforcement.

An operating system for AI-assisted work is the same idea pointed at an agent. The model brings fluency and reasoning; it cannot bring persistence (sessions end), accumulation (project state lives outside it), discipline (rules held in a prompt decay), or verification (its confidence is not evidence). The operating system is the durable structure around the agent that owns those four jobs.

One distinction holds for the rest of this book: the operating model is the way of working, portable across tools and vendors; the operating system is that model built into durable structure around a specific setup.

Mine is four layers. Each exists because one of the chapter 1 failure modes demanded it.

Layer 1: doctrine

Doctrine is the written answer to "how do we work here?": principles, conventions, role postures, and the reasoning behind them, hand-authored as plain prose in version control.

Two properties make doctrine different from a system prompt. First, it is durable and shared: it does not live inside any one tool's settings, so it survives a change of agent, machine, or vendor. It is just markdown; you can read it, a colleague could read it, any future tool can read it. Second, it carries the why. "We never offer 'do it the unsafe way anyway' as an option" is a rule; doctrine records the reasoning: a presented option is a sanctioned path, and under pressure every presented option eventually gets picked, so where no safe route exists the honest offer is a refusal. An agent that knows the why argues less and generalises better, and so does the version of you who reads it back in six months.

Doctrine answers settled questions so they stay settled. Without it, every session re-litigates decisions you have already made, one confident suggestion at a time.

Layer 2: the knowledge store

The knowledge store is accumulated project state: what happened, what was decided, what is in flight, what is deferred. Mine is a folder of markdown notes with a few conventions: a handoff note at the end of every session, one note per significant decision with its rationale, programme notes for multi-week efforts, and indexes that make the pile navigable.

The store is the layer that kills the context tax from chapter 1. A fresh session reads the latest handoff before asking me anything, so "what were we doing?" costs a file read instead of twenty minutes of my morning. The store only ever grows; it is the system's long-term memory of the project, distinct from the agent's memory of me.

That distinction matters, which is why memory is its own layer.

Layer 3: memory

Memory is the small, curated set of standing facts the agent loads about how I work: corrections that became rules, preferences that became defaults, hard-won gotchas that must never be relearned the expensive

way. The em dash rule from chapter 1 lives here. So do a few hundred others, each one paid for.

Memory differs from the knowledge store in scope and loading. The store is looked up on demand; memory's index is ambient, one line per entry, present in every session before I type a word, with the entries themselves retrieved when a line matches. That makes curation the whole game: an ambient index full of stale or wrongly-inferred descriptions decides what every session pulls in. Chapter 5 is about the difference between memory that compounds and memory that bloats, and why my system asks before it saves rather than hoarding everything it notices.

Layer 4: enforcement

Doctrine names rules. Enforcement is the machinery that catches violations before they land: checks that run automatically at the moments that matter (before a commit, before a publish, at session start), packaged workflows that make the correct path the easy path, and reviewer agents that gate specific concerns before anything ships.

This is the layer most setups skip, and its absence is why they stay stuck at advice. A rule that lives only in prose is a suggestion; under deadline pressure, agents and humans alike ignore suggestions. The principle my system encodes is that discipline exists twice: once as prose that names the rule and carries the why, once as machinery that catches the violation. When only the prose exists, the rule is not yet real.

> **Sidebar, July 2026.** In Claude Code specifically, the mapping is direct. Doctrine's entry point is the CLAUDE.md instruction files it loads at session start. Memory is the auto-memory directory. The knowledge store is any folder of markdown you point it at. Enforcement is hooks (shell commands that run automatically on events like session start or before a tool call), skills (packaged, invocable workflows), and subagents (scoped agents you delegate to, including read-only reviewers). Every layer in this chapter can be assembled from those primitives and plain files; later chapters do exactly that.

The arrival test

A folder of markdown and some scripts is not an operating system, in the same way a pile of bricks is not a house. What upgrades the pile is a small set of invariants that hold across every layer:

- **One source of truth per fact.** Every fact lives in exactly one place; anything else that states it is a derived view, regenerated from the source rather than edited by hand. Chapter 4 is entirely about this, because everything else in the system rests on it.
- **Discipline encoded twice.** Prose names the rule; machinery enforces it. One without the other is either invisible or arbitrary.
- **Portability by layer.** Doctrine, store, and memory content are plain files that survive a tool swap; only enforcement is written against a specific agent runtime, and it is the layer you expect to re-implement to attach the model to a new tool.

The test that tells you the system is working is what I think of as the arrival test. A fresh session, on a fresh morning, with no briefing from you, arrives already knowing three things: how you work, what was decided last time, and that something will catch it if it breaks a rule. When all three hold, the assistant stops being an intern with amnesia and starts being a coworker. When any of them fails, you are back in chapter 1.

What it is not

Four boundaries, each of which I have watched people (including me) get wrong.

It is not a bigger prompt. Pasting doctrine into a system prompt reproduces the words but not the properties: nothing accumulates, nothing enforces, nothing survives the session. The value is not in the text; it is in the text being durable, loaded automatically, and backed by machinery.

It is not a product you install. The shape transfers; the contents do not. Your doctrine encodes your settled arguments, your store holds your project's history, your memory is the residue of your corrections. Buying someone else's is like buying someone else's diary: coherent, and useless to you. This book hands you the shape and the discipline, on the explicit understanding that the system you end up with is yours.

It is not the tool. My system currently runs on Claude Code, and this book uses Claude Code for every worked example. But three of the four layers are plain files that would survive a move to a different agent tomorrow; the enforcement layer would need rewriting, which is exactly

why it is kept thin and separate. If your operating model cannot survive a vendor change, what you have is a vendor's operating model.

It is not automation all the way down. The system removes toil, not judgment. Deciding what ships, what a chapter of this book sounds like, what the studio does next: those gates are deliberately human, and later chapters defend why the verification gates stay human even when they are the bottleneck.

The component that did not survive

Definitions get more honest when they include what was removed.

My system used to have a fifth component: a small server that gave agents a programmatic query interface onto the knowledge store, with structured search, read, and append operations. On paper it completed the architecture, and the design was sound. My own rule for keeping components is that removing one must degrade a real workflow.

It failed that test in practice. The agent could already open and search the store directly, because the store is plain files and searching files is something coding agents are exceptionally good at. The API added a component to maintain without adding a capability the file layer did not already provide. Earlier this month the last workflow that still used it was re-pointed at direct file access; nothing degraded. The component is shelved, not deleted: the code exists, and a future need could revive it. But the running system today is the four layers above and nothing else.

I keep this story in the definition chapter for two reasons. First, it is the boundary lesson in miniature: every component must keep earning its

place against the dumbest alternative that works, and plain files are a formidable dumbest alternative. Second, it is a live demonstration of the discipline this book keeps returning to: the internal page this chapter is drawn from still describes that fifth component as current. The page was frozen five weeks before I drafted this chapter, and the system moved underneath it. The only reason this chapter is accurate is that checking sources against reality is a mandatory step in the pipeline that produces it. That discipline gets its own chapters, and one expensive war story, later in the book.

Takeaway

An operating system for AI-assisted work is four layers of durable structure around the agent: doctrine that settles how you work, a knowledge store that accumulates what happened, memory that holds what the agent has learned about you, and enforcement that makes the rules real. It passes the arrival test when a fresh session shows up knowing all of it unprompted. It is not a prompt, not a product, not the tool, and not finished when it merely contains words; and every part of it must keep earning its place.

The next chapter shows the system in motion: a real product, shipped to a real store, with the four layers doing the work this chapter just claimed they do.

Where the evidence comes from

Every book in this genre asks you to take somebody's word for something. Productivity claims arrive without logs. Case studies arrive without failures. The demo worked, once, for the person selling the course. Chapter 1 told you that confidence is not evidence when it comes from a model; the same standard has to apply to an author.

So before the principles, the provenance. Chapter 2 ended by promising the system in motion, and the motion is coming: this chapter shows what shipped, and a Part V case study replays the ship blow by blow. But a demonstration is only as good as your ability to audit the demonstrator. You have no reason yet to believe the studio is real, the receipts are honest, or the scale is more than a weekend project wearing a company name. This chapter is the disclosure statement. Read it the way you would read the methodology section of a paper: not for inspiration, but to decide how much weight the evidence can bear.

The studio

Async Digital is a one-person software studio in Cardiff. It is a limited company registered in England and Wales, with the unglamorous obligations that implies: a registered office, a privacy policy, and a data-protection registration with the ICO, the UK regulator. I am its only

engineer. Nobody reviews my code unless the machinery in this book does it, and nothing ships unless I press the button.

I put "one person" first because it is the constraint the whole book turns on. A team can paper over a leaky working environment with human redundancy: someone remembers the decision, someone else catches the regression, a third person notices the documentation went stale. Alone, you get none of that. Whatever is not written down is forgotten. Whatever is not enforced is skipped on the first busy afternoon. The four layers from chapter 2 were not an architectural hobby; they were the only way one person could run several products at once without dropping them.

The coworker

The other half of the setup is the AI coworker, and the word coworker is doing precise work. A tool gets configured. A coworker gets worked with, and the working relationship has history. Ours has been accumulating since February 2026, and none of it arrived pre-installed.

The history is made of small, specific things. Corrections that became standing rules, each with its reasoning written down, so they hold without being repeated. Scoped trust: there is a settled understanding of what it may do on its own, what it must verify before claiming, and what always waits for me, and that understanding was negotiated one incident at a time. The rule that "done" must arrive with the output that proves it exists because of incidents like the script in chapter 1 that could not fail. It runs work while I am away from the desk and compresses the state of play into a line or two when I come back; I have stopped re-explaining anything. And it does not defend its own work when the work is wrong.

In mid-July 2026 a review pass flagged a code comment it had written weeks earlier as too clever, against my standing rule that comments explain plainly; it conceded the finding with the words "a fair hit".

None of this is mysticism about the model. The model is stateless, as chapter 1 laboured to establish. The relationship lives in the layers around it: the memory that carries the corrections, the doctrine that carries the reasoning, the enforcement that applies both without asking me to remember. But the effect, day to day, is not like using a tool. Both sides behave differently because of what has passed between them, and that is the only definition of a working relationship that matters here.

What shipped

The claim from chapter 1, expanded, with dates.

Audient is on-device audio search and transcription for the Mac: it transcribes the meetings, voice memos, and talks already on a machine and makes them searchable, with nothing uploaded. It shipped as a free download with a one-time unlock at a modest price, no subscription. The pricing model matters to this story because it changed late in development, and the change that never finished propagating caused one of the review findings below.

It was submitted to the Mac App Store on 12 June 2026 and rejected on 17 June. Across the review exchange there were three findings. The subtitle used the word "Mac", which is Apple's trademark to use, not mine; that was fixed in the listing metadata. A server entitlement (a capability the app declares in its signature) had no purpose Apple could see; the answer was to explain the opt-in local integration it powers. And

a pricing question that was entirely my fault: the store metadata still described the app as paid-up-front, weeks after the model had changed to free-with-unlock, because the change had never been propagated. The replies and fixes went back within a day, and three days after the rejection the app was approved and live.

The environment earned its keep in that window twice. The release gate refused the first release-candidate cut outright because five release-blocking tickets were still open; I overrode nothing until each was triaged. And when Apple's pricing question landed, an audit of every surface that stated a price found the stale figure was systemic rather than a single wrong field: the marketing site's structured data and a draft launch email carried it too. Everything was corrected the same day, together. Chapter 19 does the full autopsy of the ship; chapter 17 covers what the estimation retros had already admitted along the way.

Iris is a small open-source Swift package for routing incoming URLs into typed SwiftUI navigation. MIT licence, first release tagged 1.0.0 in early July 2026. It matters to this book less for what it does than for what it proved: the same environment that ships private products can pass the very different hygiene bar of publishing code in the open.

A prior book exists: a Leanpub guide to SwiftUI navigation, written without any of this machinery. It is the control sample. I know what my writing pipeline felt like without an operating model, because I have run one both ways.

Behind those sit three more apps in flight and unshipped: a birthday-countdown app, a dual-timezone clock with a watch complication, and a tvOS app that teaches a card game. They get no names here. They appear in later chapters only as receipts.

A working day

"Shipped products" says nothing about how the days actually run, so here is the texture.

A session opens with one line from the machinery: how many issues accumulated since last time, and one command to review them. Continuing work costs a file read, not a briefing; the session reads the most recent handoff note and picks up the top item of its "next" list. A session is deliberately one coherent unit of work: a bug root-caused and fixed, a chapter drafted, a release step executed. On the way out, the work passes whatever specialist gates it triggered, and the session ends by writing the handoff note the next one will read. Merging, releasing, and anything else irreversible stays with me. That is not a temporary caution I expect to relax; it is a design decision the enforcement layer holds me to.

Two days from the record, to give the cadence scale. On 21 June 2026 the handoff shelf gained thirty notes across at least eight distinct workstreams, among them post-launch polish on the shipped app, three other apps, case-study articles for the package that would ship as open source a fortnight later, and maintenance of the operating system itself. On 4 July, eighteen notes across nine workstreams. One person does not do that by working faster. The cadence works because arriving in any project costs nothing: every workstream keeps its own state on disk, and any session can resume any of them cold.

> **Sidebar, July 2026.** In Claude Code terms, a "session" here is one terminal instance of the agent. The one-line greeting is a session-start hook reading buffered notices. The specialist gates are subagents: scoped, mostly read-only reviewer agents for code conventions, App Store readiness, licensing, and language quality. Twenty-one agents of all kinds are registered on this machine at today's count, July 2026; the reviewer gates are among them. Handoff notes are plain markdown in a folder; the "next session reads it" step is a skill that runs on request, not magic. Several sessions can run in parallel on different projects, which is how a thirty-handoff day happens.

The corpus

Everything above leaves a paper trail, and the paper trail is what this book is written from.

The knowledge store has been accumulating continuously since February 2026, with its earliest session record dating from the previous November. The handoff shelf passed a thousand notes in July 2026. Alongside it sit the distilled memory files and packaged workflows that chapter 1 already counted, here with their provenance: each memory file is a correction or hard-won fact that survived curation, and the bench of agents from the sidebar above polices the lot. Every factual claim in this book is backed by something in that pile: a handoff, a ticket, a post-mortem, a dated decision note. Where a number is exact, the receipt is exact; where I have rounded, it is because the true figure moves daily.

That record is not evenly spread, and the tilt is worth knowing. May was the heaviest month on the shelf at 360 notes, June ran to 301, and July,

when most of these chapters were drafted, to 218. The lessons accumulated across the whole of it; the examples nearest to hand while writing did not, because they came from the weeks I had just worked. So July is cited in these pages more often than it was worked, which is a fact about when I wrote rather than about when the system was learned. One rule follows, and it holds for the rest of the book. A date on an incident is when the incident happened. A date on a figure belongs to the figure: when the count was true, or which period it counts. A date on a tool detail is when I last checked that detail against the live system. None of the three is when the practice it sits under was learned.

The pile also records the failures, which is the part I can offer that most of this genre cannot. The same-day publish-and-revert that chapter 1 already confessed to; chapter 20 has the autopsy. A local model that was built into the pipeline and then removed, because its evaluation gate caught it inventing claims. Estimation reviews that caught the same class of miss two retros running. Even this chapter generated one. My own retrospective note about the Audient ship described the wrong platform; it had been drafted from the shape of a predecessor product. The same mandatory source-check that saved chapter 2 from its frozen page caught this one too. Part V is built from material like this. It is, I think, the strongest evidence in the book, because nobody fabricates their own post-mortems.

What this proves, and what it does not

Honesty about the sample size: this is one studio, one person, one stack, and no control group. Nothing here demonstrates that an operating model makes you faster than a colleague working without one, or that my

numbers transfer to your project. Survivorship is real, and I am the survivor telling the story.

What the record does demonstrate is existence: one person, with this structure, shipped a paid product through App Review, published open source, kept three more products moving, and produced an audit trail honest enough to include its own mistakes. When later chapters claim a practice works, the claim means: it held up here, in the day-to-day running of a real one, with receipts. You are being shown the evidence, not sold the conclusion. Judge accordingly.

Takeaway

You now have the referent for every claim that follows: one engineer in Cardiff, one AI coworker with months of shared history, a paid Mac app, an open-source package, a prior book, three apps in flight, and a written record large enough to audit and honest enough to include the failures. That is what "the studio" means for the rest of this book.

Part II starts with the idea the whole environment stands on: every fact lives in exactly one place, and everything else that states it is a copy that will eventually lie. The stale retrospective this chapter caught is a small preview of why that rule earns a chapter.

END OF THE SAMPLE

That is the sample