Contents

Agents That Earn Their Keep

Building AI Agents That Solve Real Problems, Survive Operational Constraints, and Stay Accountable

Dru Edwards

Copyright

The practices in this book come from the author's own building and operating work, told at the level his published writing already tells it. Volatile specifics (model capabilities, prices, benchmarks, tool status) are dated to the time of writing with sources in the notes; recheck them before acting. Nothing here is legal, financial, or professional advice.

DRU: add ISBN, imprint, cover credit before print

PART ONE: BEFORE YOU BUILD ANYTHING

Chapter 1: The Demo Is Not the Product

I train people on clinical systems for a living. The day job. Real people in clinical environments, high stakes, real consequences if the tool fails them. I have watched what happens when something that looked clean in a training room hits a floor nurse at 11 PM, short-staffed, three competing alerts open on her screen, trying to find a piece of information the system definitely has but that she cannot locate because the workflow made sense to the person who built it, not to the person who has to use it.

That is not a model failure. The model, the workflow, the interface, all performed exactly as designed. It is a design failure. The design was validated in conditions that did not resemble the conditions where the design would actually be used.

I go home at night and build AI agents. The tension between those two parts of my life is not lost on me.

What I have watched in the AI space is the same failure in a different costume. A demo works. Everyone sees it work. Someone builds a plan around the assumption that it will keep working. Then it meets real users, real data, real constraints, real time pressure, and the failure shows up as a model problem because the model is the visible part. The model gets the blame. The design gets rebuilt, slightly differently, and demoed again.

This book is about the part nobody shows in the demo.

The Gap Has a Name

The gap between demo performance and production performance is documented, if not widely cited in the rooms where people are planning the pilots.

The numbers that circulate land somewhere between 70 and 85 percent of AI projects that look good in a demo delivering erroneous outcomes or failing to reach durable production. The most rigorously cited primary source for this is a 2024 RAND Corporation report that puts the failure rate at more than 80 percent, twice the failure rate for conventional IT projects that do not involve AI.[1] A Gartner forecast from 2019 put a similar number in circulation, though its framing is precise: 85 percent of AI projects would deliver erroneous outcomes due to bias, misaligned algorithms, or poor implementation, a prediction about quality failure rather than deployment failure, but pointing at the same systemic gap.[2] The failure is not usually the model. The model is not dumber in production than it was in the demo. The failure is everything around the model: the data it is fed, the edge cases that were not curated out, the users who are not the patient testers from the pilot, the time pressure nobody simulated.

On the agentic side, the number I have seen cited is that 88 percent of enterprise agent pilots still fail to graduate to production. The blockers named are evaluation gaps, governance friction, and model reliability. Evaluation gaps means nobody built the system for checking whether the thing worked. Governance friction means the organization's rules about what can be automated outran the team's planning for those rules. Model reliability is the catch-all that often stands in for "we did not understand what we were building." Those three things are not model problems. They are systems problems.

The distinction matters because the fixes are different. If the problem is the model, you swap the model. If the problem is the system, you have to understand the system.

There is a particular cruelty to the evaluation gap. When you build an agent without an evaluation layer, you have no way to tell whether it is getting better or worse. You make a change. The output looks different. You have no baseline. You ship it, or you do not, based on intuition. The evaluation gap means you are flying without instruments, and instruments are exactly what make a difference when conditions are not demo-clean. Building the evaluation layer is not glamorous work. It is often less than five percent of the visible output. It is the thing that tells you whether the other ninety-five percent is going in the right direction.

What Demo Conditions Actually Are

Demo conditions are not a cheat. They are a reasonable simplification for showing a capability. The problem is when the simplification gets forgotten.

In a demo, the input is curated. Someone picked the examples. Maybe they picked examples that worked. Maybe they specifically excluded the edge cases. Often they did not consciously do either, they just showed what they were excited about, which is the thing that works cleanly.

In production, nobody curates the input. The user types what they actually need, which includes the ambiguous phrasing, the missing context, the category of question the builder did not anticipate because it came from someone using the tool for a purpose that was not the primary use case but is not unreasonable.

In a demo, failure is invisible. When something goes wrong during a demo, the demonstrator usually catches it before the audience does and redirects. Or they show the good response and do not show the bad one. This is not dishonesty, it is human presentation. But it means the audience forms an impression of a system that never fails, which is not a real system.

In a demo, the user is cooperative. They are trying to make the system work. They phrase things clearly. They pick tasks where the system will succeed. In production, users are not cooperative in this sense. They are not adversarial. They are just trying to do their job, which involves using the tool while also doing ten other things, under time pressure, without having thought carefully about how to phrase their request in a way that will maximize the model's performance.

The system that handles demo conditions and the system that handles operational conditions are related but different. Building the first one is the proof of concept. Building the second one is the product.

Why the Failure Mode Looks Like a Model Problem

The reason most agent failures show up looking like model problems is that the model is the visible surface. It is the thing that generated the wrong output. The output is wrong because the data feeding it was inconsistent. The output is wrong because the edge case was not handled. The output is wrong because the user phrased the request in a way that fell outside the implied distribution of the training examples.

The model did its job. It generated a response consistent with what it was given and what it learned to do. The fact that what it was given was bad context, or the fact that the task fell outside the distribution of things it was trained to handle well, those are not model failures. They are system design failures wearing a model costume.

I have lived this directly. I build agents that sit on private knowledge, RAG over my own vault of notes, conversational agents over personal data, running on my own hardware with safety gates in front of them. They fail in exactly the ways described above. Stale snapshots. Schema drift the second I rename a field. Two code paths reading the same field two different ways because I never wrote down what the field actually means.

The failure always looked like the agent was confused. It was not confused. It had been given inconsistent context and it answered according to what it was given. The fix was never in the model. The fix was in the data layer I had not built correctly.

That lesson took longer than it should have.

The Central Argument

Most agent failures are systems failures, not model failures.

That sentence is the book's central claim. Everything that follows is either a mechanism for how a specific type of systems failure happens, or a pattern for how to prevent it.

The reason this matters is practical. If you think you have a model problem, you look for a better model. You write better prompts. You try fine-tuning. These are legitimate solutions to model problems. They are also expensive, time-consuming, and will not fix a systems problem. The agent will produce wrong output more elegantly.

If you think you have a systems problem, you look at the data contract between the agent and its context. You look at the evaluation layer, which is probably absent. You look at the guardrails, which are probably also absent. You look at what happens when the model encounters an edge case it was not trained to handle. You look at whether the failure mode is predictable or random, because predictable failure is manageable and random failure is not.

The builders who get durable results are the ones who do not blame the model when something goes wrong. They go looking for the systems failure. They usually find it.

What This Book Covers, and What It Does Not

There are thousands of resources on how to prompt better. There are tutorials for every major model, framework, and tool in the space. Most of them show you how to build something that works in a demo.

This book is about what happens after the demo.

Part I is about thinking clearly before you build anything. How to write a spec that is actually useful. How to define what "done" means before you define the agent. How to diagnose which problem you actually have before you pick a tool to solve it. These are the decisions that determine whether the rest of the work is building toward something or building in circles.

Part II is about building right. Hallucination is an architectural property, not a quality problem. Memory is a design decision, not a feature you add. The data layer beneath the model is where most reliability failures actually live. Guardrails are not overhead. Each of these is a systems decision with a wrong approach and a right approach, and the wrong approach looks fine in a demo.

Part III is about operating honestly. Evaluation is the discipline that separates teams that ship reliable products from teams that ship demos. The cost structure of running an agent at scale is different from the cost structure of running it in a controlled experiment. The model you are building on is a dependency you do not control. These are operational realities that most guides skip, because most guides stop before the thing has to survive contact with real work.

Part IV is about building for the long game. The chatbot as a standalone product has a limited runway. Most AI wrappers fail not because the team is bad but because the capability improves and the wrapper does not have the depth to stay ahead of it. The productivity gains from AI tools do not automatically accrue to the people doing the work. These are strategic realities that show up after you have already shipped something.

This is not a table of contents recitation. It is the shape of the argument. Each part answers a question. Part I: are you thinking about the right thing? Part II: are you building the right way? Part III: are you operating honestly? Part IV: are you building for what lasts?

The answer to all four has to be yes before an agent earns its keep.

An Honest Account of Where I Stand

I want to be specific about the author's position here, because it matters for how to read this book.

I am not a researcher. I am not a founder with a funded AI company. I am a builder who works after hours, in the margins between a day job and everything else, building systems on my own hardware that are actually running, actually serving real purposes in my own work, and actually failing in the ways I describe.

The AI space is full of people who write about building agents and who have never run one in production on real work with real constraints and real people waiting on the output. I am trying to be the other thing. The tension I carry from the day job, watching what happens when technology that made sense to the builder meets the person who has to use it under time pressure, that tension is useful. It makes me ask the operational question before the demo question. It makes me think about the user who is tired and the system that was never tested with a tired user.

The other honest thing: I have been wrong about this. I have built agents that failed in exactly the ways I am describing as preventable. The data layer I skipped because it felt like extra work. The evaluation suite I did not build because I was confident the agent was working. The lineage logging I deferred because the system was small enough that I could reconstruct things by hand.

I built agents that looked good and then failed. I rebuilt them. I learned the boring fixes. The boring fixes are what this book is about.

There is also something I want to say clearly before the book gets into mechanisms and patterns. I am not a disinterested observer writing about other people's failures. Every failure mode in this book is something I have done, or something I have watched myself almost do before catching it. The data layer I describe skipping in Chapter 8 is one I actually skipped. The evaluation suite I describe not building in Chapter 14 is one I genuinely deferred. The argument here is grounded in practice, including the parts of practice that were wrong. That is why I can describe the failure modes specifically. They are not theoretical.

This also means the book has a bias. I build at the scale I can manage, on my own hardware, in the hours available to me. The systems I describe are not enterprise deployments with hundred-person teams and six-figure infrastructure budgets. They are the systems that a careful solo builder, or a small team, can actually construct and operate. The principles are the same at any scale. The specific implementation details will need to be calibrated to your context. That calibration is yours to do. The pattern recognition is what I am trying to give you.

The Named Tool: The Demo Audit

Chapter 1 does not have a complex tool. The tool is simple and the reason it is not done more often is that it is uncomfortable.

Before you build the agent for production, write down three things about the demo.

First, what inputs did you use in the demo? Be specific. List them. Now ask: what was excluded? Not intentionally excluded. Just not shown. What categories of input did not appear? What would happen if a real user submitted something in one of those categories?

Second, what did failure look like in the demo? If nothing failed, you either have a very robust system or you did not probe it hard enough. In the demo environment, run the agent against the ten inputs you most want to avoid, the ones that make you nervous, the ones that fall outside the clean examples. Write down what happens.

Third, who are your demo users? Are they the same people who will use the production system? If not, what is different about the production users? Are they under more time pressure? Do they have different technical backgrounds? Do they use different vocabulary for the things they are asking about?

Write those three things down before you build the production system. They tell you where the gap is. The gap is where the work is.

That is the move for this chapter: the demo audit. Do it before you commit to the production build. The most expensive thing in AI development is not the model cost. It is the cost of building the wrong system because you were optimizing for demo conditions.

What Survives the Demo

A system built to survive contact with real work looks different from a demo. It handles inputs that were not curated. It has a way of knowing when something has gone wrong. It fails predictably rather than randomly. It was tested by someone who was not trying to make it succeed.

Most of those things are not interesting to show in a demo. They are not exciting capabilities. They are the engineering disciplines that make a capability usable instead of impressive.

The thing I keep coming back to from years of watching technology meet real people in hard moments: a system that looks clean in a controlled setting can be brutal when a real person with a real problem and real time pressure sits down with it. The controlled setting is not the product. The person sitting down is the product. Everything you built has to answer to them, not to the demo.

That is the discipline this book is about. Not how to build the impressive demo. How to build the thing that survives after the demo ends.

Confidence: Known. The production failure rate statistics are documented in primary sources. The RAND figure is a confirmed 2024 primary source; the Gartner figure is real but framing matters. The operational framing, the systems-versus-model diagnosis, and the demo audit tool are all derived from direct practice. The observations about demo conditions are first-principles analysis that any builder who has shipped to production will recognize.

The move: before you commit to your next production build, do the demo audit. Write down what inputs you used, what failure looked like when you probed hard, and who your actual users are versus who your demo users were. The gap between those answers is the production risk.

The demo is not the product. The product is what the demo promises to people who are counting on it.

Chapter 2: Spec First, Agent Second

The fastest people I have watched work with AI agents type the least.

That took me a while to understand, because the surface-level reading is wrong. It is not that they are slow. They are moving on a different part of the problem. They spend most of their time on the thing that is not code, and that is exactly why the code part goes faster than everyone else's.

Here is the answer up front: the people getting durable results from AI agents do not open the chat and start building. They write the system down first. What it does. The rules that cannot be broken. What "done" looks like. Then, and only then, the agent starts coding.

That ordering is the whole game.

Why the Ordering Matters

When you hand an agent a half-finished spec, you do not get a half-finished result. You get a complete, faithful, confident implementation of the half-finished spec. The agent does not notice that your context file still has placeholder text from a draft you forgot to clean up. It does not quietly infer what you must have meant. It executes against what you gave it, baggage included.

I have experienced this directly. I have handed agents prompts with notes embedded, with incomplete constraints, with fuzzy success criteria written in the hope that the model would fill in the blanks charitably. It never does. It fills them in confidently, which is different. The result looks polished. The result also solves the wrong problem.

The failure is not the agent's. The agent is doing its job. The failure is in what I handed over. And a vague brief does not get better on the other side of an AI. It gets bigger.

This is the central problem the chapter addresses. Not "how do I prompt better" but "how do I think clearly enough before I prompt that the prompting is almost an afterthought."

If you have used AI agents the unstructured way, you already know the failure mode even if you have not named it. The first version sort of works. You iterate. Every pass drifts a little from the last one, because the model has nothing solid to anchor to. Six prompts later you have a codebase that carries the fingerprints of a conversation, not a design. You cannot quite explain why the auth layer looks different from the data layer. You are not sure which parts were your idea and which were the model's. The code runs. You are not confident you understand it.

Anchor the model to a written design before it writes a line, and that drift collapses. The code gets generated against a fixed reference instead of a fuzzy conversation. You can swap models, restart the session, hand the project to a different person or a version of yourself three months from now, and the work stays coherent, because what the agent reads is the spec, not the chat history.

That is the difference between speed that compounds and speed that buries you.

The Six-File Context Structure

The pattern that keeps showing up among people doing this well is a `context/` folder at the root of the project with six markdown files. Plain on purpose. The value is not that they are clever. It is that they exist, they are short, and they are true.

`project-overview.md` is what you are building, who it is for, what success looks like, and what is explicitly out of scope. One screen. No vague words like "good" or "intuitive," because those do not tell the agent anything. If your success criterion cannot be tested, even informally, rewrite it until it can.

`architecture.md` is the tech stack, the system boundaries, and the invariants that cannot be broken. "Auth always goes through Clerk." "All writes go through the repository layer." The things that should make an agent stop and ask before it touches them. If you do not write these down, the agent will make the decisions for you, and it will make them consistently, confidently, and sometimes incorrectly.

`code-standards.md` is your conventions. TypeScript strict mode, file naming, import order, the boring stuff the agent will quietly violate if you do not write it down. Every project has these conventions. In most projects they exist only in the head of whoever started the project. When an agent is the one starting units of work, there is no head to consult, so you write the file.

`ai-workflow-rules.md` is how the agents are allowed to work. One feature at a time. No drive-by refactors. Update the tracker after every unit. This is the file that keeps scope from sprawling. Without it, an agent that finishes a task and notices something adjacent that looks fixable will fix it. That is initiative in a human. In an agent it is scope creep you did not authorize and may not even notice until you are debugging something you did not ask for.

`ui-context.md` is design tokens, palette, type scale, component conventions. So the UI work matches the rest of the system instead of looking like five different agents built five different screens. On a solo project this might feel excessive. It is not. The second agent session is already effectively a different agent: different context window, different starting state. The spec is what keeps it coherent.

`progress-tracker.md` is what is done, what is in flight, what is queued. The agent reads it at the start of every unit and updates it at the end.

Then one more file, usually `AGENTS.md` or `CLAUDE.md` at the repo root, that tells every agent the same thing: read all six context files before you do anything, and update the progress tracker after each unit. Without that pointer, the context files exist but get ignored. The pointer is what makes the system actually work.

The six files are not documentation for humans. They are the agent's working memory. Without them, every session restarts from zero.

Three Things That Change When You Do This

Once you run a project this way, three concrete shifts show up.

The model stops drifting. Two different agents, or the same agent on two different days, produce coherent work because they are both reading the same architecture file. You stop re-explaining yourself. The explanation is in the file.

Hand-offs get cheap. Move the project to a different tool, a different person, a version of yourself two months from now. The context files travel with the repo. Nothing important was trapped in the chat history, which will be gone or illegible to a fresh session anyway.

Scope creep gets visible. When the only sanctioned way to do work is "pick a unit off the tracker, do it, update the tracker," then any drift from the plan becomes a deliberate move instead of an accident. You do not wake up to find half the auth layer rewritten because the agent got curious about something adjacent.

None of these shifts are theoretical. I build with agents in the loop on my own hardware, and the behavior difference is not subtle. When the agents I run have a `skills/` directory and a clearly written behavior spec, they are calmer, take fewer wrong turns, and I can restart a session without losing the thread. When I skip the spec and throw a prompt and hope, I get the drift described above. Same agent, different result. The difference is the writing I did before it ran.

The Part I Got Wrong: The Progress Tracker

I want to be specific about the piece I had not fully done myself, because it is also the piece most people skip.

I tend to keep state in my head and on scratch notes. That habit works right up until I take a break and come back. Then I spend the first fifteen minutes of the next session reconstructing where things were. The agent spends that time either waiting for direction or, worse, re-doing work I had already approved because nothing in its context told it that work was done.

When I forced myself to put state into a file the agent updates, two things happened. The agent stopped re-doing approved work. I stopped re-explaining context after every break. Both add up fast across a multi-session project.

That is not a theory I read. That is the thing I most recently got wrong and fixed.

The progress tracker feels like overhead because you have to update it. It is not overhead. It is the thing that prevents the overhead of re-explaining yourself every time you come back to the project. You are not adding work, you are shifting it to a place where it compounds instead of disappearing.

Most people skip the tracker because it feels redundant, the work is already in their head. It stops feeling redundant the first time you come back from a three-day interruption, read the tracker, and know exactly what state the project is in, what the agent last finished, and what comes next.

The Failure Modes Are Predictable

Four things go wrong consistently, and they all trace to the same root: someone built the context folder as a one-time setup and then let it drift from reality.

Writing the files once and forgetting them. Stale specs are worse than no specs, because now the agent trusts something that is wrong. The fix is treating the context folder like code. Update it whenever the system's real shape changes, before you build the next thing. When the architecture changes, you update `architecture.md`. When a new convention gets established, you update `code-standards.md`. The spec has to be the source of truth, not a historical document.

Vague success criteria. "Make it good" cannot be checked. Every line in `project-overview.md` should be testable, even informally: "users can sign up and create up to five projects without paying." If your definition of done is not specific enough for you to verify it yourself, the agent cannot verify it either.

Skipping the tracker because it feels redundant. It is the file that stops you and the agent from re-doing work. The habit is: every unit starts with "read tracker" and ends with "update tracker." Build that habit before you trust it to scale.

Letting agents touch architecture. Implementation is agent territory. Architecture is yours. Anything that would change `architecture.md` needs a human edit first, and then the agent implements the consequences. The moment you let an agent make architectural decisions, you have ceded the decisions that determine everything about the system's long-term health. The agent will make those decisions helpfully and confidently. It does not carry the cost of living with them.

What a Vague Spec Produces

It is worth being precise about what "vague spec" actually does to agentic output, because the failure is not obvious until you have seen it a few times.

A vague spec does not produce garbage. It produces something that looks complete. The agent fills every gap with its best inference about what you wanted, based on patterns from everything it has ever processed. Those inferences are often reasonable. They are not what you wanted. And they are delivered with the same confidence as every other output, so there is no signal that the gap was filled rather than specified.

You review the output and it looks like code. You might even run it and it works. What you have built is a faithful implementation of an underspecified idea, which is a different thing from what you meant to build. The delta between those two things is the tech debt you will pay later, when the third session diverges from the second because neither had a stable reference.

This is why the spec discipline matters even for small projects. The smaller the project, the more tempting it is to skip it. You know what you are building. You can hold it in your head. The problem is the agent cannot hold it in your head. It can only read what you wrote down.

A fuzzy conversation is not a spec. A system prompt is not a spec. A good spec is a written design that an agent could implement without asking you a single follow-up question. If you could not hand it to a new person and have them start contributing, it is not done.

The Spec as Working Memory

The framing that made this click for me was thinking about what the spec actually is in a multi-session, multi-agent project.

It is not documentation. Documentation is written after the thing exists, to explain it to people who did not build it. The spec is written before, to constrain the thing that is about to be built. Those are different documents with different jobs.

The spec is the working memory of the whole system. Every agent that touches the project reads it. Every session starts from it. It is the shared ground truth that keeps six different agent interactions from drifting in six different directions.

Human working memory is fast, flexible, and unreliable across a session break. Written specs are slower to create, rigid until deliberately changed, and perfectly reliable across breaks. The tradeoff is obvious once you have experienced the cost of an agent that had to guess what you meant and guessed wrong.

Most AI coding tools, Claude Code, Cursor, Aider, automatically pick up an `AGENTS.md` or `CLAUDE.md` at the repo root. This is not a coincidence. It is the tooling acknowledging that the spec needs to be the first thing the agent reads. Use it. Put your six context files there and a pointer to read them. The cost is one paragraph. The leverage is high.

The Named Tool: The Six-File Spec Template

This chapter's named tool is the spec template. Not a worksheet, not a framework, just the six files and the principle behind each one.

The principle is: write down everything the agent would need to know to do this unit of work without asking you a follow-up question. If there is something it would reasonably ask, write the answer in the file before it asks.

Here is how to set it up.

Create a `context/` folder at the root of the project. One sitting. Keep the files short. The value is not in their length, it is in their existence and their accuracy.

Write `project-overview.md` first. What are you building, who is it for, what does success look like in concrete terms, and what is explicitly out of scope. Out of scope matters. An agent without a scope boundary is an agent that can wander.

Write `architecture.md` second. Tech stack. System boundaries. The invariants. When in doubt, add a line. An explicit invariant you write down and later change is less expensive than an implicit one the agent violated because you never stated it.

Write `code-standards.md` third. Conventions. File naming. Import order. TypeScript configuration. The stuff that makes code reviewable by a human who was not there when it was generated.

Write `ai-workflow-rules.md` fourth. How agents are permitted to work on this project. One feature at a time. No architecture changes without a human edit first. Update the tracker before ending a session.

Write `ui-context.md` fifth. If there is no UI in the project, make the file and note that. A deliberate "no UI" is not the same as a forgotten file.

Create `progress-tracker.md` sixth. Start it with the first unit of work. Format: task name, status, brief notes. The agent updates it at the end of every unit. You update it when a unit is approved or when the plan changes.

Add `AGENTS.md` or `CLAUDE.md` at the repo root. One paragraph: read the six context files before starting any work. Update the progress tracker at the end of every unit. That is the whole thing.

Then treat the spec like code. It is the artifact that everything else depends on. When the system changes, the spec changes first, and then the agent implements the new spec. Not the other way around.

The Counterweight

The spec-first discipline has a version of itself that becomes its own failure mode.

You can spend so long writing the spec that you never build anything. The spec becomes a comfort object, a substitute for the harder work of implementing and learning what you got wrong. This is less common than under-speccing, but it happens, especially in people who have been burned by vague-spec failures and overcorrect.

The spec needs to be true and useful, not complete and comprehensive. You do not have to resolve every edge case in the spec before you build the first unit. You have to resolve the invariants, the scope, the success criteria, and the conventions. The rest emerges from implementation and gets written back into the spec when it does.

The goal is a spec that takes one sitting to write, not one week. If it takes longer than that, the scope is too large. Break it into pieces, each with its own spec. Build in small enough chunks that the spec can be accurate rather than aspirational.

A stale spec is worse than no spec. A useful spec is accurate, short, and updated when reality changes.

Asking the Right Question Before You Open the Tool

There is a question worth asking before you open any AI tool to start a project.

If a new person joined this project tomorrow, could they read six files and start contributing?

If the answer is no, the spec is not done yet. If you do not even know what is missing, that is the most important signal: the project's rules exist only in your head, and your head is not available to an agent, only to you.

The fastest way to write code with AI is to not write code with AI right away. Write down what you are building, what the rules are, what is out of scope, and what done looks like. Then let the agent execute against that.

The people getting durably good results are not the ones who found the smartest model. They are the ones who treat the agent as the cheapest, fastest part of the loop, and put their own thinking upstream of it, in files that survive the session.

Confidence: Known. Every observation in this chapter traces directly to experience running multi-session, multi-agent projects on my own hardware. The six-file structure is a pattern I have described publicly and used in practice. The progress-tracker confession is recent and specific.

The move: before you open your next AI session, spend thirty minutes writing or updating the context folder. Specifically, write a `progress-tracker.md` if you do not have one.

The spec is not the output. The spec is what makes the output worth keeping.