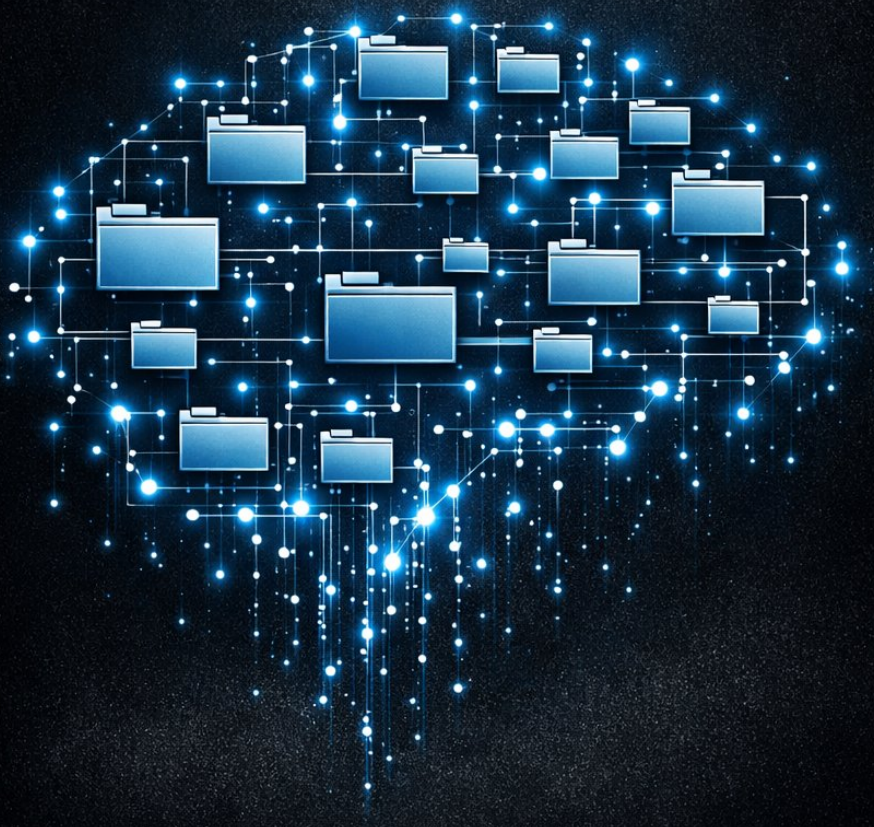
PROJECT BRAIN

How to stop your AI from forgetting your projects



Slawomir Luzny

Founder, FixFlex LTD

PROJECT BRAIN

How to stop your AI from forgetting your projects

Slawomir Luzny — Founder, FixFlex LTD

Why this book (one paragraph)

This is not the story of how I got here — that's another book. This is the method: a small, boring, plain-text convention that made AI agents stop forgetting my projects, stop mixing them up, and stop burning tokens re-reading the same documentation every session. No SaaS, no database, no framework. A folder of Markdown and a few rules. It grew out of a real problem in real products — and it works across more than one tool. That's the whole pitch.

TABLE OF CONTENTS

PART I — THE PROBLEM

1. **AI Has Amnesia** — the brilliant consultant who forgets everything overnight. Why it gets worse over time, not better.
2. **The Real Cost Is Re-Loading, Not Generating** — the expensive part isn't writing code, it's restoring context every session.
3. **Why README Stops Working** — one big always-loaded document gets heavier week by week; the model drowns in it.
4. **The Multi-Project Trap** — where memory rots into wrong answers: a fact from project A glued onto project B.

PART II — WHERE THE IDEA CAME FROM

1. **It Started With Sentinel and Queen** — the mechanism was borrowed, not invented: agents that report what they did and check it off.
2. **From Server Fleet to Project Memory** — taking "report-and-acknowledge" from infrastructure and pointing it at project knowledge.

PART III — THE ARCHITECTURE

1. **Index First** — the model reads one small map before anything else.
2. **Small Index, Details on Demand** — cold storage: only the index loads eagerly; topic files load when a question needs them.
3. **Status Carries the Outcome** — verified / done / in-progress / failed / superseded. The difference between "works" and "we tried that and it broke."
4. **Provenance: Knowing What You Don't Know** — human-confirmed vs. AI-inferred, and why the model should weigh them differently.
5. **Staleness: Memory With an Expiry Date** — facts age; flag them for re-confirmation instead of trusting last year's note.
6. **The Guardian of Structure** — a tiny zero-dependency validator that keeps the map honest.

PART IV — THE METHOD IN PRACTICE

1. **First Run and the Daily Start** — init light; one brain per server; the morning handover.
2. **The Discipline of Saving** — suggest, don't auto-dump; a human decides what's worth remembering.
3. **Session Handover** — closing a session so the next one picks up cleanly.

4. **One Brain, Many Tools** — the same brain across different agents, and the honest limit.
5. **Keeping Secrets Out** — a brain holds architecture and variable names, never secret values.

PART V — TOKEN ECONOMICS

1. **Token Bankruptcy** — the wall where the limit ends exactly when you need it.
2. **Why Agents Read Too Much** — and how a map stops the reading-just-in-case.
3. **Measuring the Saving Honestly** — when the token win is real and when it's modest.

PART VI — THE FUTURE

1. **A Standard, Not a Product** — why this stays free and open, and why that's the strength.
2. **One Brain for Any Agent** — what portability really requires.
3. **Memory Belongs to the Project** — not to the vendor, not to the tool.

PART VII — PHILOSOPHY

1. **Open Source vs. Vendor Lock-In**
2. **The Operator, Not the Coder**

CHAPTER 1 — AI HAS AMNESIA

There's a common misunderstanding about artificial intelligence: that it remembers everything. It doesn't.

Most modern models behave like a brilliant consultant with short-term memory loss. Inside a single session, the consultant is extraordinary. It reads thousands of lines of code, understands the dependencies, catches bugs you'd have missed, sketches an architecture in seconds. For a few hours, it feels like you've hired the best engineer you've ever worked with.

Then the session ends. And most of that understanding is gone.

The next morning you start again from almost nothing.

At first this costs you nothing, because there's nothing to forget. A small project has a handful of endpoints, one database, a few screens. The model picks up the whole picture in a minute or two, and you barely notice the gap.

The trouble starts a few months in.

The project has grown. There are dozens of tables now. Hundreds of commits. New servers. Integrations that each have their own quirks. A changelog. A roadmap. A

trail of decisions — this was tried and abandoned, that was chosen and why. The thing has a history.

And every new session, the model knows none of it.

So you start each day the same way. Read the README. Read the changelog. Read the architecture notes. Read the roadmap. The model dutifully consumes thousands of tokens just to arrive back at the place it was yesterday — before you've asked it to do a single useful thing.

It's like walking a new hire through the entire history of the company every single morning, from the day it was founded, before they're allowed to touch any work.

After a while I noticed something absurd about my own days. I was spending more energy teaching the AI my project than building it. The tool that was supposed to make me faster had quietly added a tax to every session: the cost of remembering.

And here's the part that took me too long to see. This was never a problem with the model's intelligence. The model was fine. It was sharp every single time.

The problem was memory. Specifically, the absence of any memory that belonged to the *project* instead of the *conversation*. When the conversation ended, the knowledge died with it — even though the project, sitting right there on disk, hadn't changed at all.

That's the gap this book is about. Not making the AI smarter. Making the project remember more than the people and the tools that touch it — so that no session ever has to start from zero again.

The fix, when I finally found it, was almost embarrassingly simple. It wasn't a database. It wasn't a service. It wasn't another AI wrapped around the first one. It was a folder of plain text, arranged so the model could find its way without reading everything — and a short list of rules about what goes in it and what never should.

But before the solution, it's worth sitting with the problem a little longer — because the shape of the problem is exactly what the solution has to fit. And the next, sharper version of the problem isn't forgetting one project. It's quietly confusing two.

CHAPTER 2 — THE REAL COST IS RE-LOADING, NOT GENERATING

When people think about the cost of working with AI, they think about generation. About what it costs to write a function, generate a test, sketch a component. That's the visible part — you watch the model spit out code, and you know you're paying for it.

But that's not where the real cost is.

The real cost is invisible, because it happens before you ask for anything useful at all. It's the cost of loading. The cost of getting the model to the point where it understands what you're working on in the first place.

Think about how the start of a typical session on a mature project looks. You open a new conversation. The model is empty — sharp, but empty. So you start loading it. You paste the README, because otherwise it doesn't know what the application is. You paste the directory structure, because otherwise it guesses where things live. You paste the database schema, because without it the model invents tables that don't exist. You paste the changelog, the architecture notes, the recent decisions.

And only now, after consuming thousands of tokens, is the model ready. You're at the point you were yesterday at this same hour — before you started any real work.

That's the tax. You pay it every morning, before you build anything.

On a small project this tax is so low you don't feel it. The context is a few files; the model swallows them in a second. But the cost of loading doesn't grow linearly with the project — it grows with every thing the model "has to know" so it doesn't guess. The more mature the project, the more such things there are. More tables. More decisions to remember. More traps you don't want it stepping into again.

At some point the proportion flips. You start the day and realize that most of the session — and most of your limit — went on simply preparing the model, not on work. Generating code was cheap. What was expensive was getting to the point where that generation made sense.

And here's the insight that changes how you think: if the biggest cost is loading, then loading is what you optimize — not generation.

It's not about the model writing less code. It's about it reaching a full understanding of the project while reading as little as possible. So that instead of consuming the company's entire history every morning, it reaches for the one right page — the one it actually needs.

That is exactly what I later designed Project Brain to do. But before we get there, there's one more reason "dump

everything in the README and let it read" stops working — and it isn't about cost. It's that the more the model reads at once, the worse it gets, not better.

CHAPTER 3 — WHY README STOPS WORKING

The first instinct of anyone who notices this problem is the obvious fix: if the model forgets, let's write everything down in one place and make it read that on startup. One big file. All the project knowledge. A README that knows everything.

For a while it works. Then it starts to break — and for two reasons pulling in opposite directions.

The first is weight. A file like that never gets thinner. Every new decision, every integration, every fixed pitfall adds another paragraph. Week by week it gets heavier. And since the model is supposed to read it at the start of every session, every week you pay more for the same thing — back to the tax from the previous chapter, except now it grows on its own. The file that was meant to save time starts eating it.

The second reason is worse, because it isn't about money — it's about quality. The more you give the model at once, the more it drowns in it. It's a paradox that's easy to miss: more context doesn't mean better answers. Past a point it means worse ones. The model gets a thousand lines at once and stops distinguishing what matters for the current question from background

noise. Everything has the same priority, so nothing does. It starts mixing details, joining things that shouldn't be joined, losing the thread.

Now add a second project. And a third. When all your knowledge sits mixed in one big, always-loaded document, an overloaded model starts taking facts from one project and applying them to another — and it does it so smoothly you only catch it when something fails to work. That's the most dangerous version of this problem, and the next chapter is devoted to it.

So the problem with the "big README" isn't that it's a bad idea. It's everyone's natural first idea. The problem is that it scales in exactly the wrong direction: the more you know about your projects, the more mature they are, the heavier the file is to load and the easier it is for the model to get lost in it. It grows with your success and punishes you for it.

What you actually need is something that works the opposite way. Something where the model reads very little at the start — just enough to know what exists and where to find it — and reaches for detail only when a specific question demands it. A small map instead of a thick book. Loading bounded by an index, not by the whole history.

And that's the point where we stop talking about the problem and start talking about the solution. Because

that's exactly how Project Brain is built — and, interestingly, the idea itself didn't come from thinking about AI. It came from something that was already running on my servers.

CHAPTER 4 — THE MULTI-PROJECT TRAP

So far we've talked about one project. The model forgets it, expensively re-remembers it, and a big README makes that worse, not better. But there's a version of this problem that's more dangerous than all the previous ones combined — and it shows up exactly when things start going well for you.

Because success in this work doesn't look like one project. It looks like five. Six. You have a product that earns, a second that's still growing, a third stood up for a client, plus a couple of tools that run all the rest. Each has its own stack. Its own database. Its own decisions and its own pitfalls. And they all live in your head at once — because you know which is which.

The model doesn't.

And that's where the trap begins. When the knowledge about all your projects lies mixed together — either in one big book or just in your successive explanations pasted into the chat — the model will at some point take a fact from one project and apply it to another. This one's on FastAPI and Postgres. That one's on Node and MySQL. And the model, overloaded with context at three in the morning, hands you a solution tailored for

the second one while you're working on the first. It writes a query against a table that doesn't exist in this project. It proposes a pattern that fit somewhere else. It refers to an endpoint from the neighboring product.

And now the most important sentence of this chapter: it sounds convincing.

Because the model always sounds convincing. It doesn't say "I'm not sure which project this is." It doesn't stop. It hands you the wrong answer with the same confidence as the right one. The syntax checks out, the tone checks out, the structure looks professional. Everything looks fine — except that it's working off someone else's map.

That's exactly what makes this trap the most dangerous. Ordinary forgetting is visible. You ask the model something it doesn't know, and it either guesses out loud or asks for context — either way you know the gap exists. Confusing projects is invisible. You get no warning. You get a smooth, complete, wrong answer that looks exactly like a correct one. You catch it only when something breaks — or, worse, when it works in the wrong place.

I remember it from my own work. A content generator for one of my sites started describing infrastructure that site didn't have — because it "knew" about it from another project. It invented metrics. It quoted a client

number nobody had confirmed. It described a database architecture that existed elsewhere. None of it was malicious or stupid — they were real facts, just from the wrong project. The model wasn't lying. It simply reached into the wrong drawer, because all the drawers were lying open on one table.

And that's where you arrive at the conclusion behind all of Project Brain: the problem isn't that the model knows too little. The problem is that it has no way to know which thing belongs to which project. What it lacks isn't knowledge — it's boundaries.

A big README won't fix this — it makes it worse, because it throws everything onto one table. Keeping separate files helps a little, but only as long as the model itself remembers which file it's using — and an overloaded one stops. What you really need is for the structure of the memory itself to enforce the boundary. For every fact to carry its project, and for the system to make sure a fact from project A doesn't masquerade as a fact from project B. For confusing the drawers to be not so much "discouraged" as technically caught.

When I was building Project Brain, that was one of the first mechanisms I judged indispensable — a guard that checks whether a note sits in the right project's folder, and warns when a topic from one project name-drops another without explicitly declaring it. Not because it sounded clever. Because I'd stepped into this trap

myself and knew what a smooth, convincing, wrong answer costs.

That's where the description of the problem ends. We now have the full picture: the model forgets, expensively re-remembers, drowns in too much context, and confuses projects without letting it show. Four faces of one absence — the absence of a memory that belongs to the project, not to the conversation.

The rest of this book is about the solution. And the solution, perhaps surprisingly, wasn't born from thinking about artificial intelligence. It was born on my servers — out of two systems that had long been doing something I only later recognized as memory.

CHAPTER 5 — IT STARTED WITH SENTINEL AND QUEEN

I said at the start that the solution wasn't born from thinking about artificial intelligence. That's true — and it's worth saying where it really came from, because it changes how you look at it. Project Brain wasn't an idea at a desk. It was a pattern that had long been running on my servers — just pointed at something completely different.

To explain it, I have to show two of my systems. Not to brag — to show the mechanism.

The first is Sentinel. It's my tool that sits on a server and watches over it: the databases, the security, the performance. When something's wrong — a slow query, an attack, a strange service state — Sentinel detects it and, when it's safe, fixes it itself. You can put it on many servers at once. Each such instance is a separate guard, working in the field.

The second is Queen. And this is where the part that matters begins.

Picture a beehive. The Sentinels are the worker bees — scattered, each on its own server, each seeing only its own slice of the world. Queen is the queen: one, at the center, hidden. No user has access to her — only me.

The workers don't chatter with the queen non-stop, don't bombard her with questions. They do something else: when they hit a problem, they send the queen a report. Short, structured. What happened, in what environment, what the symptoms were, what was already tried.

The queen collects these reports from the whole swarm. And every so often — not immediately, but in batches — she sits over them and processes them. From many reports she builds what I called the Knowledge Book: a set of patterns. "This symptom, in this environment, usually means this, and this is what worked on it." The Book is shared by the whole hive — every worker can draw on it, not just the one that reported the problem.

And now the most important detail, the one that later became the heart of Project Brain. Every pattern in the Knowledge Book carries counters beside it: how many times a given solution helped, how many times it didn't. When a worker applies advice from the Book and reports back whether it worked, the queen checks off the outcome — bumps the success or failure counter and records when it was last confirmed. Knowledge in the hive isn't faith. It's a history of what actually worked in the field — with a number beside each entry.

I worked with this system every day. I watched the swarm report, the queen build the Book, the patterns earn their "worked eighteen times, failed twice." And at

some point — on completely different work, fighting with the fact that AI kept forgetting my projects — a thought arrived that felt almost too simple.

I already have a memory that works.

My hive had long been doing exactly what I lacked in my work with AI: it kept knowledge outside a single conversation, in a structured form, marked with whether something was confirmed or not, and made it available to anyone who needed it. Except that memory was aimed at server security. And I needed exactly the same thing — just aimed at project knowledge.

I didn't have to invent anything from scratch. I had to take a mechanism I already knew and turn it in a different direction.

A "drone report" became an entry about a project. The "Knowledge Book" became a map of projects the AI reads at the start. The "worked / didn't work" counters became a status that carries an outcome: verified, done, failed. The rule "the swarm doesn't chatter with the queen, it feeds a shared library" became the rule that memory belongs to the project, not to a single session with the model. Even the fact that the queen works in batches and with deliberation, rather than reacting to every twitch — that came back later as the rule that you save to the brain consciously, rather than dumping everything in.

That's the whole origin. Not an epiphany, not a startup, not a framework. A pattern from the hive, carried over to a developer's desk. Something that had guarded my servers for years turned out to be exactly what my work with AI needed — the moment I looked at it from the right angle.

In the next chapter I'll show what that leap looked like in practice — what had to change when memory from a server fleet was to start serving not failures, but knowledge about projects. Because while the mechanism was the same, the goal placed entirely new demands on it.

CHAPTER 6 — FROM SERVER FLEET TO PROJECT MEMORY

The mechanism was the same. But as I started turning it from guarding servers to remembering projects, it turned out the goal placed entirely different demands on it. Three things had to change — and each teaches something about how project memory differs from fleet telemetry.

The first change: who decides what gets saved

In the hive, saving is automatic. A drone detects a problem and reports — on its own, without asking. And rightly so, because there completeness is what matters: you want every anomaly to reach the queen, because you never know which report will form part of an important pattern. The automatic nature is a virtue.

For project memory, that same automation would be a disaster.

If the brain saved everything that happened in a session — every attempt, every dead end, every thought said out loud — it would quickly turn into a swamp. The map that was meant to be small and readable would swell to the size of that same big document we fled from in the first part. The more the automation dumped in, the less the map would be worth.

So here I reversed the rule. In the brain, saving isn't automatic — it's a conscious decision. At most the system can remind you: "you did a piece of work and the map didn't change — maybe worth saving?" But it writes nothing on its own. It can't and isn't allowed to. The human decides what's worth remembering and what was just an attempt along the way. Because project memory isn't meant to be a complete record of everything that happened. It's meant to be a short list of the things that actually matter. And that takes judgment an automaton doesn't have.

That was the first lesson: in telemetry, more means better. In memory, less means better.

The second change: what actually gets saved

The hive holds failure patterns. Symptom, environment, what worked, how many times. It's knowledge about how things break and how they're fixed — numerical by nature, statistical, repeatable.

Project memory is about something else. It's not about failures. It's about decisions. What the stack is and why exactly that one. What the deployment looks like. What's already been done and whether it works. What was tried and why it was dropped. Where the pitfalls are that you don't want to step into twice. These aren't metrics — they're the history of thinking about the project. The reason behind every "this is how it's done."

That difference changes what's even worth keeping. In the hive you save what happened. In the brain you save what you decided and why — because that "why" is what vanishes fastest when a session ends, and what's most expensive to reconstruct from scratch. You can read the code. The reason you wrote it that way and not another — you can't, if you wrote it down nowhere.

The third change: where this memory lives

The hive keeps its knowledge in a database. It makes sense — it's a central system, one queen, data flowing in from the whole fleet, processed in batches. A database is the right tool.

For project memory I chose something deliberately simpler: plain text files on disk. Markdown. No database, no server, nothing you have to start up and maintain.

I did it consciously, for two reasons at once.

First: text files can be read by anyone — both a human and any agent. You open a file and you see what's in it. You don't need a tool to look into your own memory. And since it's plain text, it isn't read by just one particular tool — it's read by any agent that can open a file. Memory stops being locked inside one program.

Second: since these are files, they live with the project. They go into the repository, they travel with the code,

they grow alongside it. Memory doesn't live in a separate system you have to wire up to the project — it lives in the project itself. That's exactly the difference the rest of this book is about: memory belongs to the project, not to the tool and not to a separate database off to the side.

In the hive a database was a virtue, because the knowledge was shared across the fleet and centrally processed. In a project the virtue is the absence of a database — because the knowledge is meant to be local, portable, and readable with the naked eye.

What came out of it

Three changes, one thought underneath. Fleet telemetry wants to be complete, numerical, and centralized. Project memory wants to be lean, decision-based, and local. The same mechanism — report, keep status, check off what's confirmed — but turned so it serves not a queen gathering data from a swarm, but one person who wants their project to remember more than a single session with the AI.

Once those three decisions were made, one question remained, the simplest and the hardest at once: how do you actually arrange these files so the model reads little and knows a lot? Because swapping a database for Markdown solves nothing if that Markdown is, again,

one big file. That — the architecture of the map itself — is what the next part is about.

CHAPTER 8 — INDEX FIRST

The whole first part of this book came down to one tension. The model has to know a lot about the project to be useful. But the more it reads at the start, the more it costs and the easier it is for it to get lost. Knowledge and cost pull in opposite directions.

A big README tries to solve this by force: give the model everything, let it cope. We saw why that fails — the file swells, the cost grows, the model drowns. It's an attempt to win by quantity.

Project Brain solves it the opposite way — by order. Instead of asking "how much should the model read," it asks "in what order." And the answer is: very little first, then exactly as much as needed.

At the heart of it is one small file. The index. The map.

The index holds no detail. It holds a listing. What projects exist. In each of them — what topics. And beside each topic — its status and a pointer to where the full content lives. That's all. The index doesn't say *how* the cache problem was solved. It only says: "a cache problem exists, it belongs to this project, it's solved and verified, and the details are in this file."

It works exactly like a table of contents in a thick book. You don't read the whole book to find out whether it has

a chapter on something. You look at the table of contents — one page — and you know what's inside and on which page. The brain's index is the table of contents of your projects.

The model reads that listing at the very start of the session. In a few seconds, at the cost of a handful of tokens, it knows something priceless: what exists, in which project, and in what state. It doesn't know the details yet — but it knows where to look for them when they're needed. That's a completely different starting point than "an empty, brilliant consultant." It's a consultant who got the company map before they started.

And here's a subtlety that's easy to miss, and is the most important. The knowledge of "what exists and in what state" alone — with no details — already solves two of the four problems from the first part of the book.

It solves confusing projects: since the index says outright that this topic belongs to project A, the model won't glue it onto project B. The boundary is on the map.

And it solves quietly repeating work: since the index says the logo was swapped and verified three days ago, the model won't quietly do it a second time. Instead it tells you it's already done and asks whether you want to repeat it or do something new. The map knows what's

finished — so the model stops treading on its own tracks.

All of that from one small file, read at startup. Without touching a single detail. The index isn't a summary of the knowledge — it's a map to it. And as you'll see in a moment, that distinction is what makes the whole system cheap.

CHAPTER 9 — SMALL INDEX, DETAILS ON DEMAND

The index tells the model what exists. But at some point the model needs the specifics — not "the cache problem is solved," but "how exactly it was solved." And here comes the second half of the idea, the one that makes the whole thing cheap: the details are loaded only when a specific question requires them.

The full content of each topic lies in a separate file. How it was solved, what was tried earlier, why the approach was changed. These files are cold storage — they lie on disk and nobody reads them until they're needed. The model doesn't load them at the start. It doesn't load them "just in case." It reaches for them only when a question leads specifically to them.

Back to the cache example. You ask the model: "how did we solve the cache problem?" The model doesn't read the whole knowledge base. It reads the index — the one it's had since startup — and sees: cache, such-and-such project, verified, details here. It follows that one pointer, opens one file, and answers. It read exactly as much as the question needed — one topic. It didn't touch anything more.

That's the whole difference in cost. With a big README the model reads everything to answer anything. Here it reads the map plus one topic. And — most importantly — that cost doesn't change as the project grows.

Stop on that, because it's the most important sentence about the brain's architecture. In an ordinary notes file, every new thing you write adds to what the model reads every time. The file swells, the cost grows week by week. In the brain, new topics add to cold storage — to files the model doesn't read anyway until asked about them. The amount of knowledge you hold grows. The amount the model loads at startup doesn't. The session cost is bounded by the size of the index, not by the size of the whole history.

That means you can keep six months of decisions in the brain, dozens of topics, the project's entire history — and the session still starts as cheaply as on day one. Because at startup the model reads only the table of contents, not the whole library. The library can grow without limit. The table of contents stays short.

And that's why the brain doesn't get fat over time, while an ordinary notes file does. It isn't a matter of discipline in writing briefly. It follows from the shape itself: the hot part (the index) is small by design, and everything that grows lands in the cold part, which nobody reads without a reason. The project's success — having an ever-longer history — stops being a punishment. In the

big-README model, the more you knew, the more expensive it was to begin. Here you can know any amount, and the start costs the same.

A small map, always read. Details, read on demand. Two rules, and from them follow both the low cost and the resistance to swelling. The rest of the architecture — statuses, provenance, freshness — are layers laid on this foundation. But the foundation is these two things. Start from the map. Reach for the details when a question demands them.

In the next chapter we add the first of those layers — something that separates a project map from an ordinary "done / not done" list. Because "done" isn't enough. You have to know whether what's done *works*.

CHAPTER 10 — STATUS CARRIES THE OUTCOME

An ordinary to-do list knows two states: done and not done. You check an item off and move on. That's enough for a shopping list. For a project — no.

Because "done" says nothing about the thing that matters most: whether it works.

Picture two entries, both checked off as "done." The first: we added an index on this table and the queries sped up — verified, works. The second: we tried setting up the cache this way and it crashed, we dropped the idea. Both are "done" in the sense of "we already dealt with this." But they mean opposite things. One is a solution. The other is a warning.

An ordinary list flattens them. A model that sees only "done" doesn't know whether to repeat it or steer well clear. And if it guesses wrong — it either tears down something that worked, or steps a second time into the same thing that already failed.

So in the brain, status isn't binary. Status carries the outcome. Every topic on the map is marked not by whether it was dealt with, but by how it ended:

- **verified** — done and confirmed to work.

- **done** — finished, but not yet confirmed in the field.
- **in-progress** — started, not yet finished.
- **failed** — tried, didn't work. That's not an absence of work — it's the knowledge that this road doesn't lead through.
- **superseded** — once current, now a different approach; kept so the trail survives.

The difference between "verified" and "failed" is everything to the model. With "verified" it knows this is a foundation it can build on, and that there's no reason to touch it. With "failed" it knows this is a dead end — and instead of proposing the same solution that already fell over, it avoids it. Status doesn't just say "we were here." It says "we were here and this is what came of it."

That's exactly the same pattern you saw in the hive — where every entry in the Knowledge Book carried a number: how many times it helped, how many it didn't. There it was a history of field-tested solutions. Here it's narrower and simpler: one outcome marker beside each topic. But the thought underneath is the same — memory without an outcome is half a memory. Knowing something was tried is little. What matters is whether it worked.

And one more thing follows from "superseded." When an approach changes, the old one doesn't vanish — it's

marked as superseded and lies there still. Because what was tried earlier and why it was dropped is itself knowledge. Six months from now, when someone (or you) asks "maybe do it like this?", the map answers: "tried, here's why it was changed." The brain doesn't overwrite history. It versions it.

CHAPTER 11 — PROVENANCE: KNOWING WHAT YOU DON'T KNOW

There's a quiet trap in any memory you keep alongside an AI. When you write something down in black and white, that something hardens over time into a fact — regardless of where it came from.

Picture two sentences in the project notes. The first: "the database runs on port 5432" — because you checked it yourself and wrote it down. The second: "the database probably runs on port 5432" — because the model guessed that during a conversation and someone pasted it into the notes. After a week both look the same: they sit in a file as recorded knowledge. After a month nobody remembers that one was a fact and the other a guess. The guess hardened into certainty — just because it was written down.

That's a real risk when an AI co-authors your memory. The model can write something convincingly, and a convincing record reads later like truth. An ordinary notes file protects nothing here — it blurs the difference between "a human confirmed this" and "the model made this up."

The brain keeps that difference explicit. Every note can carry a provenance marker: whether a human

confirmed it, or the model wrote it without confirmation. It's a tiny label, and it changes everything — because it separates what you know from what you only suspect.

And here you have to be precise, because this works on two levels that are easy to confuse.

The first level: the validator. There's a small checker that makes sure the provenance field exists at all and has a sensible value — "human" or "ai-inferred." The validator doesn't judge whether the note is true. It only checks that the label is in place and well-formed. That's a purely mechanical check of form.

The second level: the weighing of trust itself. That happens when the model reads the brain. Seeing that something is human-confirmed, it treats it as certain. Seeing that the model merely inferred something, it double-checks before building on it. That's not a function of the validator — it's a behavioral rule, an instruction for the model on how to read the markers. The validator guarantees the label is there. The model — that it's taken into account.

I'm being this precise because it's exactly the kind of precision missing from most promises around AI. It's easy to say "the system distinguishes facts from guesses." The truth is narrower and more honest: one part makes sure provenance is recorded, and another

makes sure it's taken into account when reading. Those are two different things in two different places. Together they give something rare: a memory that knows what it doesn't know.

CHAPTER 12 — STALENESS: MEMORY WITH AN EXPIRY DATE

Even a true fact has a shelf life. What's certain today is sometimes out of date in six months — not because someone lied, but because the world moved. Keys get rotated. Versions go up. "Current" stops being current. And a recorded note doesn't know that. It lies in the file with the same certainty as the day it was written — even if the world has long overtaken it.

That's the third trap of memory, after confusing projects and after guesses hardening into facts. An old, once-true record, served today as still true. The worst kind of error, because it looks credible — because it *was* true.

The brain handles this by giving facts an expiry date. A topic can carry a deadline: how long we consider it fresh. And if nobody set one, there's a default horizon — roughly six months. When a topic passes its deadline, the validator flags it: "this was confirmed long ago, re-confirm before relying on it."

Notice what this does, and what it doesn't. It doesn't delete old knowledge. It doesn't guess whether a fact is still true. It does something humbler and wiser: it raises its hand and says "this has aged — check it." It leaves the decision to the human. Because the system doesn't

know whether the port changed. It only knows a lot of time has passed since anyone confirmed it — and that's enough to warn, instead of serving last year's note as gospel.

That's the same philosophy again as with provenance. The memory doesn't pretend to omniscience. It doesn't claim to know what's current. Instead it honestly marks the limits of its own certainty: this is fresh, that needs re-checking, this other thing was only a guess. A wise memory isn't one that knows everything. It's one that knows what it's no longer sure of.

And here the whole part closes. The map says what exists. The details wait on demand. Status carries the outcome. Provenance separates knowledge from guess. Freshness makes sure yesterday's truth doesn't pose as today's. Five simple ideas, none complicated on its own — and together they add up to something an ordinary notes file doesn't give: a map that is not only a record, but its own honest reviewer.

One question remains: who makes sure this structure doesn't fall apart? Because a map with a broken pointer, or a status that lies, is worse than no map at all. That — the small guardian that keeps it all in line — is what the next chapter is about.

CHAPTER 13 — THE GUARDIAN OF STRUCTURE

Everything I've described so far rests on one quiet assumption: that the map is true. That the pointer leads where it should. That the status in the index matches the state of the topic. That a note lies in the right project's folder. The whole value of the brain stands on being able to trust it — and a map that lies is worse than no map. Because when you have no map, you're careful. When you have a map that leads to the wrong place, you walk straight into the error, trusting it.

So someone has to make sure the structure doesn't drift. That someone is a small checker — in the brain it's called brain-check.

It isn't artificial intelligence. It isn't anything complicated. It's a simple program, with no external dependencies, that you run whenever you want, and that walks through the brain checking the boring but critical things:

Whether every pointer in the index leads to a file that actually exists — or points into the void. Whether the topic headers are well-formed. Whether the status in the index matches the status in the topic itself — whether the map doesn't say "done" while the topic says

"in-progress." Whether a note isn't filed under the wrong project. Whether a fact hasn't passed its freshness deadline. Whether the provenance label has a sensible value.

This is hygiene. Just as in code you run a linter to catch dumb errors before they get expensive, here you run brain-check to catch a drifting map before the model falls over it. Boring, mechanical, reliable — and valuable for exactly that reason.

But now the thing I have to say outright, because it's the difference between an honest description of a tool and a sales pitch.

The validator checks structure and freshness. And only that.

It doesn't check whether your knowledge is true. It doesn't judge whether a note marked "verified" actually works — it only makes sure the label is in place and well-formed. Remember the two levels of provenance from the previous part: the validator guarantees the provenance field exists and has a sensible value — but whether the model *weighs* that trust when reading is the model's behavior, not the checker's function. Same with the save reminder I'll come to later — that happens during work, it isn't a validator check.

I say this sharply because it would be easy to write "brain-check makes sure your memory is true and

trustworthy" — and it would sound great, and it would be a lie. The validator doesn't know whether something is true. It only knows whether the structure holds: pointers lead where they should, statuses match, labels are in place, deadlines haven't passed. That's a lot — because a drifting structure is the most common way a map starts to lie. But it's not the same as "guards the truth." And I'd rather tell you exactly what it does than promise more than I deliver.

That's a trait of the whole brain, by the way, not just the validator. Every element does one simple thing well and doesn't pretend to do more. The index maps. The topics hold the details. Status carries the outcome. Provenance separates knowledge from guess. Freshness watches the age. The validator checks that it all holds together. None of these things is clever on its own. The cleverness is that together they give a map you can trust — and one that tells you itself when you can't.

That's where the architecture ends. You now know what the brain is built from and why exactly so. It's time to show what it looks like in real work — not in theory, but on my own projects, which I maintain with this method every day. That's what the next part is about.

CHAPTER 14 — FIRST RUN AND THE DAILY START

Theory is fine, but one thing matters: what it looks like when you sit down to work. So let me show how the brain works for me day to day — because I maintain many projects at once with it, on several servers, through Claude Code.

First, the honest order of things. My applications — the security tools, the content automators, the platforms — came into being long before Project Brain. The brain wasn't first. It came later, as memory added to projects that had been running for a long time. And it came for a very simple, personal reason: I was tired of constantly reminding the model what we'd actually worked on last time. I'd sit down to work and every time I'd reconstruct the same background. At some point I asked myself a question that felt almost naive: how is it possible that this doesn't remember what it should? The brain was the answer to that one question.

The first run of the brain is deliberately light. It doesn't read all your code, doesn't analyze every line. It looks around at the signals that lie on the surface anyway — config files, repositories, the directory structure — and from that it builds a small index: what projects are here. That's cheap and fast. You don't pay at the start for

combing through all the code, because at this stage it's only about the brain knowing what exists at all. The rest fills in on its own, as you work.

And here's an important detail of my setup — because I don't have one giant brain for everything. I have several, distributed. The rule is simple: one brain sits on one instance. The brain lives on a server, beside the projects that live on it. When I connect to a given server through Claude Code, the model gets access to that server's brain — the one that knows those particular projects. Another server has its own brain, with its own map. Memory isn't centralized in one place — it lives where the project it concerns lives.

That makes sense when you run things spread across several machines. Each server carries the memory of its own projects locally, beside them. There's no single sack everything falls into. There are several maps, each in its place, each about its own slice of the world. And since the brain is just files lying on the server, the memory is exactly where it's needed, not in a separate system off to the side.

But the real value shows only at the daily start — and that's the moment I built this for in the first place.

The start of every session used to look the same: I'd open Claude Code and start explaining. What the project is, what it runs on, what I did last. Now it's

different. I open the session, and the model — instead of being an empty genius — first reads the map. And it knows right away. It knows what we were working on last. It knows what I saved at the end of the previous session as "to do next time." It knows what state the project is in.

Instead of me explaining to it, it tells me: last time we did this and that, it was saved that the next thing to handle is this — shall we continue? That's a reversal of roles. I don't begin the day by reconstructing context in its head. I begin with it already having it and asking where we start. It finds its bearings immediately. Those few minutes of explaining that used to open every session — gone.

It sounds small. A few minutes. But multiply it by every session, every day, across many projects — and suddenly it's a real part of the day that I got back. Not to mention I also spare myself: I don't have to remember where I left off at one in the morning. The map remembers for me.

CHAPTER 15 — THE DISCIPLINE OF SAVING

Since the brain remembers for me, a question arises: how does it know what to remember? And here returns the rule I described at the origin — the one thing I deliberately reversed relative to the hive. The brain doesn't save on its own. Saving is a decision.

That's crucial, so I'll put it another way: the brain can remind me it's worth saving something, but it never writes anything of its own will. After a piece of work it can point out — "you changed a lot and the map didn't move, maybe worth recording?" But I decide whether that was something worth remembering, or just an attempt along the way that shouldn't clutter the map.

Why does this matter so much? Because if the brain saved everything automatically, it would quickly become the thing I fled from — a big, cluttered document where you can't find what matters. Every blind attempt, every momentary state, every thought said out loud would land on the map. And a map that has everything on it is just as useless as a map that has nothing.

So the discipline of saving isn't a limitation — it's what keeps the map valuable. I save when a piece of work is finished and I know whether it worked. I save a decision

and its reason. I don't save every step that led to it. The map is meant to be a short list of the things that matter — and it stays short only when someone with judgment decides what goes on it.

That's a human's job, not a machine's. The machine doesn't know that a three-hour fight with a bug ended in one sentence worth saving, and the rest were dead ends to forget. I know. And that's why the last step — "this is worth remembering, that isn't" — stays with me.

CHAPTER 16 — SESSION HANDOVER

There's one thing I do at the end of work that closes the whole loop: I record where I stopped and what's next. It's like leaving a note for my future self — or for the next session, which might even be in a different tool.

Because sessions end — sometimes because I finished the work, and sometimes because the limit said stop, or it was simply one in the morning and I had to go to sleep. And it's exactly that moment, breaking off work halfway, that's most dangerous for memory. The next day you don't remember exactly where you stood. What worked and what you hadn't checked yet. What the next step was that you had in your head before you closed the laptop.

So before closing, I leave a short trail in the brain: did this and that, checked or not, next time handle this. That's a shift handover — exactly like on a construction site, where the crew going off tells the crew coming on what's done and what's left. Except here I hand the shift over to myself, for tomorrow.

And that's exactly what I see the next day at the start I wrote about earlier. The brain greets me with that trail: last time this, saved that, to do this. The loop closes.

The end of one session is the beginning of the next — with no gap, no rebuilding from zero, no "where did I actually stop?"

It may be the simplest habit in this whole method, and it gives the most peace of mind. I can close the laptop halfway through the work, at any hour, and know that tomorrow I won't start by hunting for my own tracks. I'll start from where I stopped. Project memory isn't there to remember everything. It's there so you never start from zero.

CHAPTER 17 — ONE BRAIN, MANY TOOLS

There's one more thing I didn't expect when I built this — and it turned out to be one of the strongest sides of the whole idea. Since the brain is plain files on disk, it isn't tied to one tool. Any agent that can open a file can read it.

I tested this for real, not in theory — on one server, one brain, several different tools, in separate sessions.

Claude Code went through the full loop: it read the brain, added new topics with the correct structure, ran the validator. Windsurf — the same, in full, and what's more: the saved state survived a restart. I closed it, opened it again, the brain was in place. A third tool, from an entirely different vendor, connected over SSH and correctly read the brain, pulling the right project details on demand.

That means one brain can serve several tools at once. I save something in one agent, and the next — even a different one — picks up the same map, the same statuses, the same history. Memory stops being locked inside one program. It belongs to the project, and the tools merely draw on it.

But I have to be honest about the limit — because this is exactly the kind of thing that's easy to overstate, and I'd rather tell the truth than a promise.

It works when the tool is driven by a sufficiently strong model that can really use tools — read files, act, write. I tried connecting a small, local model. And it failed. Not because the brain was bad — the brain is the same files. It failed because the model was too weak to go through the whole read–use–write loop. Worse, instead of admitting it couldn't read a file, it started inventing its contents. Three times it gave a different, false version of the same brain.

That was an important lesson for me, and I'll leave it here as such: reading the brain is nearly universal — almost any tool will manage. But fully managing it — reliable reading, conscious saving, validation — requires a model that can really use tools. The convention is portable. The quality depends on the engine that reads it. Memory is only as good as the model that reaches for it.

In the course of those tests I ran into something worth describing — because it shows why an orderly brain beats what tools do on their own. Remember how I have my memory laid out: one brain per server, each knowing only its own projects. Testing the brain on one of the other tools, I noticed it started mixing projects from different servers — not just from the one it was actually

connected to. That puzzled me. How does it even know about projects from another machine?

The answer turned out simple and instructive. These tools save their history not only on the server side, but also locally, on my computer. And from that local, private memory they were pulling context — mixing into the current session scraps from completely different projects that had once passed through them. Their own memory was spilled across the laptop and knew none of the boundaries the brain enforces.

That's exactly the contrast that makes the brain valuable. The brain is clean and ordered: one brain per server, every fact assigned to a project, boundaries on the map. A tool's native memory can be the opposite — spilled across the machine, without clear boundaries, mixing what shouldn't be mixed. And one more thing, worth knowing regardless of project memory: since tools keep history locally, it's worth checking where exactly, and what ends up there — because sensitive things can land in such logs, things you'd rather know are written to disk. It's not about being afraid. It's about knowing — and that, too, is part of caring for your own memory.

That's a good place to close the part on practice honestly. The brain isn't magic. It's a simple convention that works wonderfully with a strong tool and poorly with a weak one — exactly like any good tool. It doesn't

promise to work everywhere the same. It promises that where you have a capable model, it gives it a memory it wouldn't otherwise have.

CHAPTER 18 — KEEPING SECRETS OUT

There's one rule in running a brain that I'd write in capital letters if books allowed it. The brain holds knowledge about how things are built — never the secrets that protect them.

It sounds obvious. Everyone would sign off on it. And I still broke that rule in my own brain — in several places at once. I'll tell you about it, not to flog myself, but because it's exactly the mistake anyone will make who runs such a memory late at night, tired, in a hurry. Better you hear about it now, from me, than discover it in your own setup after a leak.

First, the distinction everything stands on. There's a huge difference between writing down a name and writing down a value.

"The database key sits in a variable with such-and-such name, in the config file" — that's knowledge about architecture. It says where to look for something, how the system is laid out. It gives away nothing that anyone with server access wouldn't already know. Such a note belongs in the brain. It's exactly what the brain is for.

"The database key is this string of characters" — that's a secret value. And it must never end up in the brain.

Because at that moment your project map stops being a map and becomes a vault — a file that hands real access to anyone who reads it.

That boundary is the whole rule: a variable name — yes. A path to a config file — yes. The value itself, a password, a token, a key — never. Values live in config files out of the brain's reach, in a password manager, in environment variables. Not on the map.

And now why this is more dangerous than it seems — and why good intentions alone aren't enough.

The first reason: the brain is files, and files travel. They go into the repository. They get copied between machines. If a secret value once lands in the brain, it doesn't politely stay in one place — it rides everywhere the brain goes. One careless moment and a secret that was meant to lie in one protected file is suddenly in several copies, in history, maybe on its way somewhere public.

The second reason, the one I discovered the hard way — back to the previous chapter. The brain is read by many tools. And tools save what they read, locally, on their own side. So a secret written into the brain doesn't even stay in the brain. It scatters into the local logs of every tool that opened that brain. One write in one place — and copies land in several, beyond your control. And that's the moment when "but it's my private server"

stops being enough, because the secret hasn't sat only on that server for a long time.

In my case it came out during ordinary cleanup — I ran through the brain looking for things that shouldn't be there, and found several. Passwords whose values I'd once written in "for convenience." Keys that were meant only to be described, but were given in full. None of it was malicious or stupid at the moment of writing — each crept in because at one in the morning it's easier to write the value outright than "see the config." But the sum of those small conveniences was a real hole.

And here's the second lesson, more important than the first. A rule in your head isn't enough. I knew you mustn't keep secrets in the brain — and I kept them, because a tired person breaks their own rules, even the ones they believe in. What actually works isn't a resolution. It's a barrier you put in the machine, not in your head. A validator that scans the brain and shouts when it finds something that looks like a secret. A habit of combing through the memory now and then. A tool that makes sure you don't push a secret where it shouldn't go — no matter how tired you are.

Because the truth is: a good system doesn't assume the human will always be vigilant. It assumes they'll be tired, and puts a net under the place where it's easiest to fall. The rule "no secrets in the brain" is right. But

only when you back it with something that works when you don't — does it become true.

And one more thing, at the end, because it's part of the same lesson. When you do find a secret where it shouldn't be, removing it from the file is only half the work. The other half is changing the secret itself — because once it leaked, even just into your own logs, the old value is burned. A clean file with a dead password protects no one if the live password still opens the door. Cleanup and rotation are two different acts. The first saves appearances. The second saves you.

That's where I close the part on practice. The brain in daily work is savings, peace of mind, and a memory that doesn't vanish at midnight. But you run it like an adult: deciding consciously what to save, and firmly watching what to never save. The next part is about something that ran through from the start, and now gets its own place — about how much all this really costs in tokens, and when the saving is real and when it only sounds good.

CHAPTER 19 — TOKEN BANKRUPTCY

Anyone who's worked with AI on a serious project knows that wall. It isn't a wall of an error or a crash. It's the wall of the limit — and it has a nasty habit of showing up at exactly the worst moment.

You're sitting over something critical. Production is down, the bug won't yield, you're in the middle of solving it. And right then the counter runs out. The daily limit, the weekly one — exhausted. Not when you were fiddling with something trivial. When you most needed the tool to keep working.

I call it token bankruptcy — because that's exactly how it works. Tokens are the currency of this work. Every file read, every model answer, every analysis costs something. And like any currency, they can run out. And when they run out in the middle of critical work, you have two options, both bad: either you top up in a hurry, under pressure, just to finish — or you close the laptop with the work half-done and the hope that tomorrow goes faster.

We spent the first part of this book showing where those tokens run off to. Not on generating code — on loading context. On the model understanding the project from

scratch every time before it does anything. That's the quiet tax: you pay currency just to get the model to the starting point.

Now connect that to bankruptcy. Since every session starts by burning tokens on loading context, that means part of your limit vanishes before you even start working. The bigger the project, the more there is to load, the larger the slice of the daily currency goes on "remind yourself where we are." Bankruptcy comes sooner, because you start each day already a little poorer.

And this is where the brain comes in not as a convenience, but as a saving of real currency. If the session start costs you a fraction of what it cost before — because the model reads a small map instead of the whole history — that means more of the daily limit is left for work. The bankruptcy wall moves further out. Not because you got more tokens. Because you stopped wasting them on the same loading, day after day.

But — and here I have to be honest, because tokens are the easiest thing to tell tall tales about — that "if" matters. Not everyone gains the same. In the next two chapters I'll lay it out precisely: why agents read too much in the first place, and when the brain's saving is real and large, and when it's only modest. Because promising you "you'll save a ton" would be easy. Telling

you the truth — when yes and when no — is more honest.

CHAPTER 20 — WHY AGENTS READ TOO MUCH

To understand where the brain saves, you first have to see why an agent reads more than it must in the first place. Because it isn't a flaw of a particular tool. It follows from the very situation the model is in at the start.

Imagine you walk into a project you know nothing about, and someone tells you to fix something in it. What do you do? You read. A lot. You open the README, you browse the structure, you look into the config — because you don't yet know what's important and what isn't. You read just in case, because it's better to know too much than too little. Only once you've grasped the whole picture do you start acting deliberately.

The model does exactly the same — except on every session, from scratch. It has no idea what's relevant to your question, so it pulls in everything it suspects might be useful. It reads just in case. And "just in case," in the world of tokens, means "expensive." Every line read pre-emptively, just in case, is spent currency — often on something that had nothing to do with your question.

Worse: the more material you give it, the more it reads just in case, because it has more to go through. A big notes file isn't only big in itself — it also encourages the model to consume the whole thing, because how is it to know that this one section at the end is the one that's needed. You pay for reading everything so the model can find that one thing.

The brain defuses this through the order that the whole of Part III was about. The model doesn't get all the material at the start. It gets a map — a small one. From the map it knows what exists and where it lies, so it doesn't have to read just in case to get oriented. When a specific question comes, it goes straight to the one right place, instead of combing through everything in search. Reading "just in case" turns into reading "exactly what's needed."

That's the whole saving in one sentence: the agent stops reading just in case, because the map tells it where to go. It doesn't read less because it was forbidden to. It reads less because it knows where to look — and someone who knows where to look doesn't have to browse everything.

CHAPTER 21 — MEASURING THE SAVING HONESTLY

Now the hardest chapter of this part — because in it I'm to tell you not what sounds best, but what's true. There's so much exaggeration floating around about token savings that adding to it wouldn't be my style. So I'll say it outright: when the brain saves a lot, and when little.

Let's start with the case where the gain is real and large. If today you keep all the knowledge about a project in one big document that the model loads at the start of every session — then the brain will change a lot for you. Breaking that into a small map plus details on demand genuinely cuts the cost of each session, because you stop loading the whole history to ask one question. The bigger that big document of yours was, the bigger the saving. There's no exaggeration here — it simply follows from reading a map instead of a library.

And now the case where the gain is modest — and I won't pass over it in silence. If you don't keep a big always-loaded document, if your project is small or you feed the model context sparingly anyway, then the token saving from the brain will be moderate. There's no fooling yourself: the brain won't conjure savings where there was no waste. If you already read little, then

reading a map instead of little will change little in the bill.

I say this outright because this is exactly the moment where it's easy to lie. I could write "the brain saves X percent" and give a number that looks nice on a cover. But that number depends entirely on how you worked before — and I don't know your "before." Any hard savings figure without that context is invented. So I won't give one.

And one more honesty, because saving tokens isn't the only — or even the most important — reason to use the brain. Even in the case where the token gain is modest, two things remain that the brain always gives, regardless of how you counted before: fewer hallucinations and a memory that doesn't vanish. The model stops inventing your deployment and confusing projects, because it has the map as an anchor. And you come back to a project after three months, and it still knows how it works. These are values you don't measure in tokens — and often it's they, not the saving, that are the real reason.

So if I had to weigh it honestly: count on token savings when you're escaping a big always-loaded document — there it's real and large. In other cases, treat it as a nice bonus, not the main reason. The main reason is that your project starts to remember — and that the model stops guessing where it should know. That you get

always. Currency savings vary. The peace of memory doesn't.

That's how I close the part on economics: not with a promise of percentages, but with a map of what to really expect. The next part looks further ahead — what happens when this same memory stops being your private one and becomes something any agent, any tool, can draw on. About the future of memory that belongs to the project, not to the vendor.

CHAPTER 22 — A STANDARD, NOT A PRODUCT

There's a question that sooner or later comes up with any tool someone finds useful: "when will you start charging for this?" And with Project Brain my answer is deliberate, considered, and may seem illogical: I don't intend to. The brain is and will stay free, open, free for anyone to take.

It sounds strange from someone who builds and sells software. I have products I monetize, and I'm not ashamed of it. But the brain is something else — and that's exactly why it's meant to be free. I'll explain, because it isn't a gesture, it's a strategy.

Project memory isn't a product. It's a convention. A way of arranging text files so the model finds its bearings in them. The value of the brain doesn't lie in some secret code I could sell — it lies in the idea itself, in how things are laid out. And an idea that's meant to catch on has to be open. A standard works only when everyone can use it without asking permission and without paying a toll.

Think of the things that genuinely spread in the world of programming. A file format. A way of laying out directories. A naming convention. None of them won because someone was selling it. They won because they

were open, simple, and anyone could adopt them in five minutes. If project memory were to become something common — something you just do, the way you keep a README in a repository — then it can't be locked behind a fee. It has to be so free there's no reason not to use it.

There's also a plain honesty in it about what the brain is. It's Markdown files on your disk and a small checker with no dependencies. There's no server I'd have to maintain, no service it would be reasonable to charge a subscription for. Taking money for a convention of arranging files would be selling air. The value goes to you, at your place, on your disk — and that's how it stays.

And the benefit for me? There is one, just different from money. A tool that's free and good builds trust. It shows I understand the problem well enough to solve it simply. That's reputation, not revenue — and in the long game it's sometimes worth more. But even if it weren't, the decision stays the same: a standard, not a product. Because only as a standard does project memory have a chance to become something you use by reflex, rather than something you buy.

CHAPTER 23 — ONE BRAIN FOR ANY AGENT

Since the brain is a standard and not a product, something concrete follows from it — something I already started telling at the practice part, and which closes here into a principle. Since it's plain text files, arranged according to an open convention, it isn't tied to any particular tool. Any agent that can read files can use it.

This isn't theory — I tested it on several different tools, as mentioned earlier. But it's worth seeing why it works, because it follows directly from what the brain is. There's no special format in it that only one tool understands. There's no API you have to plug into. There's text, arranged to be readable — for a human and for a machine. And text is read by everyone.

That means memory stops being a prisoner of the tool. In the normal arrangement, when your work history sits inside a particular program, you're tied to that program — change the tool and you lose the memory. With the brain it's the reverse. You can work with one agent today, another tomorrow, and connect a third over a remote link the day after — and they all read the same map, the same statuses, the same history. The brain lies separately, independent of whoever reaches for it.

Here, though, the limit has to be repeated, because without it this would sound like a promise with no backing. "Any agent" means "any agent with a capable model." Merely reading the brain is simple — almost anything that can open a file will manage. But fully using it — reliable reading, conscious saving, checking the structure — requires a model that can really use tools. I learned this when a weak model not only couldn't manage, but started inventing content it couldn't read. The convention is for everyone. The full extent of its possibilities — for those whose engine is strong enough.

But the direction is clear and important. The more tools can read files — and all the serious ones can — the more the brain becomes a shared language of memory between them. You don't have to choose one agent forever and pray it survives. You choose the brain as the place where knowledge lives, and you change agents like tools in a box — because the memory stayed in its place, independent of each of them.

CHAPTER 24 — MEMORY BELONGS TO THE PROJECT

We arrive at the sentence that is the heart of this whole book — and which only now, after everything that came before, can sound out in full. Memory belongs to the project. Not to the tool. Not to the vendor. Not to a single conversation that ends and vanishes.

Think about where the knowledge of your project lives today when you work with AI. In the conversation — which dies when you close the session. In the head of the tool — which keeps it for itself, in its own way, and gives it back to you only while you're using that particular one. In your own memory — which is tired, unreliable, and doesn't remember what you settled at one in the morning three weeks ago. In all those places the knowledge is someone's property — the session's, the tool's, the vendor's, your tired mind's. In none of them does it belong to the project itself.

The brain reverses that arrangement. The knowledge lands in files that lie with the project. They go with it into the repository. They stay when you change tools. They persist when the session ends. They're readable by you and by any agent you choose to connect. Memory stops being something you borrow from a tool for the

duration of use — it becomes part of the project, just like the code, like the README, like the commit history.

That's a deeper change than it looks. Because when memory belongs to the project, you stop being hostage to anything. You don't depend on whether a given tool survives, changes its pricing, or keeps supporting you. You don't lose the history when you move to something new. The project carries its knowledge itself, with it, regardless of who works on it and with what. That's freedom — the same freedom the last part of this book is about: freedom from being locked into one vendor.

And there's a simplicity in it that I like. The best solutions usually aren't the cleverest — they're the most obvious, once you see them. Knowledge about a project should live with the project. Not in the vendor's cloud, not in a separate system's database, not in a fleeting conversation. With the project, in files anyone can open. So obvious it's strange we don't all do it already. The brain is just a way to begin.

The last part is about why this one decision — that memory belongs to the project, not the vendor — is more important than it seems. About openness versus lock-in. And about who you become when you stop memorizing syntax and start thinking like someone who directs.

CHAPTER 25 — OPEN SOURCE VS. VENDOR LOCK-IN

This whole book, though it pretends to be a story about memory, is really about one thing: who holds the keys. Whether your work belongs to you, or is rented from someone who can change the terms at any moment.

Lock-in — being bound to a vendor — is the oldest trick in this industry. It always works the same way. You get a tool that's convenient, cheap to start, pleasant to use. You go in deep. You build your work, your habits, your knowledge on it. And once you're inside, once leaving would cost you too much — the terms change. The price goes up. The feature you relied on disappears behind a higher subscription. Your data is in a format you can't extract. You're not a customer. You're caught.

I've seen it up close and felt it in my own wallet. Tools that tempt with a low threshold and then tighten the noose exactly when you need them most. It isn't an accident or the malice of a single company — it's a business model. Convenience at the start, a trap at the end.

Project memory is exactly the place where this trap hurts most. Because memory isn't one feature you can quickly replace. It's the whole history of your project —

decisions, pitfalls, what works and what to avoid. If that history sits locked inside a tool, then you're bound to that tool for life and death. Changing vendors means losing the memory. And nobody wants to start from zero, so you stay — and pay whatever they ask.

That's why the brain is what it is: plain text, open, lying at your place. It isn't a random technical decision. It's a conscious choice on the side of freedom. Your memory in Markdown files you can open, read, move yourself, is a memory nobody can cut you off from. No pricing will change. No company will go under taking it with them. No format will close. It's yours, because it lies at your place, in a format anyone can read.

Open source here isn't a fashion or an ideology. It's a practical answer to a concrete question: what happens to my work if the tool I use disappears or changes the rules? With memory locked in a vendor, the answer is: you lose it. With open memory lying at your place: nothing happens, you take another tool and work on. That's the whole difference between being an owner and being a hostage.

And it's also the reason I give the brain away for free, though I sell other things. Because if I closed it and made people pay for it, I'd become the very thing I warn against. I'd create another trap. And the whole point of the brain is the opposite: to give people a memory that nobody — including me — can take from them.

CHAPTER 26 — THE OPERATOR, NOT THE CODER

At the end, the most personal thing — because this whole method grew not out of theory, but out of who I am and where I came from.

I didn't start as a programmer. For most of my life I worked physically, with my hands. I have no years of computer science studies behind me, I didn't cram language syntax, I don't know a thousand commands by heart. And for a long time I thought that excluded me from this world — that to build serious things you have to be what I'm not.

I was wrong. And the mistake turned out to be the most important lesson.

Because working with AI revealed something the old school didn't want to admit: that building software was never really about memorizing syntax. Syntax was scaffolding, mastering the tool. The essence was always something else — understanding how things should work. How they connect. What matters and what doesn't. What decision to make when there are two roads. You don't cram that. You either understand it or you don't.

When the model takes the syntax onto itself — writes the code, remembers the commands, knows every library — what's left for the human is exactly that other part. Direction. Judgment. Decision. Architecture. You stop being someone who hammers out code and become someone who directs. An operator, not a coder. And it turns out this role needs no diploma — it needs understanding, logic, and the ability to make decisions. Things you acquire through life, not only through school.

Project Brain is, in a sense, an operator's tool. Because an operator doesn't remember everything themselves — an operator makes sure the system remembers for them, so they can focus on decisions. The brain is that system. It holds the knowledge, the statuses, the history, the boundaries — so the human doesn't have to, and so the model doesn't guess. It lets one person run things that would once have required a team — not because that person knows more, but because they have a memory that doesn't fail and a model that does the dirty work under their direction.

And that's the thought I want to leave you with at the end. It doesn't matter where you come from. It doesn't matter whether you hold a diploma or a screwdriver. It doesn't matter whether you ever wrote code or put up walls. What matters is whether you can understand the problem, impose your own terms on technology, and

make decisions when the machine waits for direction. The rest — syntax, commands, the memory of details — is no longer your worry. That's what the tools are for. That's what the brain is for.

For a long time the world told people like me that this isn't for us. That without the right path, the right paper, the right past, there's nothing here for you. This book, and the method it describes, are proof that it isn't true. Memory can be given to the project. Syntax can be handed to the machine. And the place that's left for the human — the place of the one who understands and decides — was free the whole time. You only had to reach for it.

The End.