The Project Your AI Never Forgets

Why your AI costs five times more than it should — and how a handful of plain text files fixes it

One index.
One file.
Done.

The core idea comes from **Jake Van Clief** and his Interpretable Context Methodology.
Free. Sharing explicitly encouraged. · ebook.quizsparkhub.com

Read this page first

You do not have to build anything.

At the back of this book there is a prompt. You paste it into Claude, Codex, or whatever you use, answer three questions, and twenty minutes later the whole system exists. You do not need to understand a single command to get that far.

So why a book?

Because a system you do not understand is a system you will quietly break. You will move a file, delete something that looks useless, ignore a warning — and six weeks later nothing works and you have no idea why.

This book exists so you understand what your AI built for you, and why every piece is shaped the way it is.

Every part of this system exists because something went wrong first. I will show you what went wrong. That is the honest way to explain a design — not "here are the rules", but "here is the accident that made this rule necessary".

This book follows the actual story — where the idea came from, what happened when it met a real system, and only then how the result works.

Part	What it does
1	The problem, and what it actually costs you
2	Where this started — Jake Van Clief's ICM
3	What happened when it met a running system
4	Follow one question through the finished system
5	Why each piece is exactly the way it is
6	What happens when your project grows
7	Setting it up by hand — *optional, only if you want to*
8	Connecting your AI
Appendix	The prompts. This is what most people will use

Parts 1 to 6 are the book. Part 7 exists so that nothing is a black box — but you can skip it and lose nothing.

Everything here comes from one real project: a piece of software called QuizSpark Hub, built almost entirely by AI. The numbers are measured, not estimated. The mistakes are real, and several of them are mine.

And the core idea is not mine. It comes from Jake Van Clief. Part 2 is about him and his method, and it is there because you cannot understand the rest without it — everything that follows is either his idea, or something that had to be added when his idea met a system that changes underneath you.

Part 1 — The problem

1. The assistant with no memory

Imagine hiring a brilliant assistant. Genuinely brilliant — fast, knowledgeable, never tired.

One catch: every morning they have forgotten everything.

So every day you explain again. What the project is. How it works. What you already tried. What broke last time. Why that one strange thing is the way it is. By the time you finish, half the day is gone. Then they do twenty minutes of excellent work and go home.

Tomorrow, you explain it all again.

That is what working with an AI is like. Not because it is stupid — because it starts every conversation with an empty head.

So people do the obvious thing. They paste in everything: the whole project, all the notes, the entire history. And it works. It also costs a fortune, and it quietly gets worse as the project grows.

2. What it actually costs

Almost nobody measures this. Here is what happens when you do.

An AI reads text in small pieces called **tokens**. Roughly, a token is three quarters of a word. You pay per token, and there is a ceiling — the *context window* — on how much it can hold at once.

In the real project behind this book, there was a rules file the AI had to read before it could do anything at all. It was 36 kilobytes. About **10,000 tokens**. Every session. Every question. Before a single useful word came out.

And what did that file do?

It told the AI **which other file to read**. That was its whole job. A signpost.

The actual answer — the file explaining how the thing worked — was 3 to 4 kilobytes. About 1,000 tokens.

> **Nine out of every ten tokens went to the signpost, not the answer.**

Ninety percent of the cost of every question was navigation. Not thinking. Not answering. Just working out where to look.

Once you have seen that number, the entire design of this system follows from it.

3. Why "just paste everything" fails

Three reasons, and all three get worse as you grow.

It costs more every week. Your project grows, the paste grows, the bill grows. What you get out of it does not grow at all.

It runs out. Every AI has a ceiling. Once your project is bigger than the window, "paste everything" is no longer available. Now you must choose what to leave out — and you have no method for choosing, so you choose badly.

It makes answers worse. This is the part people find hardest to believe. Buried in a mountain of text, an AI is *more* likely to surface something outdated — something true six months ago, something in a note you forgot to delete.

More context is not more accuracy. Past a point it is less. You are not helping it think; you are giving it more chances to trip.

Part 2 — Where this started

None of this began with a plan. It began with a YouTube video.

The video

"Stop Building AI Agents. Use This Folder System Instead." by **Jake Van Clief**.

The title is the argument. At the time, everyone building with AI was assembling agents — frameworks, orchestration code, tools calling tools. His claim was that most of that is unnecessary, and that the thing you actually need is a folder structure.

His method is called **ICM — Interpretable Context Methodology**. The core idea fits in one line:

> **Folder structure replaces orchestration.**

Instead of writing code that tells an AI what to do and when, you arrange folders and markdown files — and the arrangement itself is the instruction. No framework. No orchestration layer. The filesystem is the architecture.

The part that makes it work

His routing has **three layers**, and each answers exactly one question:

Layer	File	Question
L0	root `CLAUDE.md`	Which workspace handles this job?
L1	workspace `CLAUDE.md`	What files does this workspace need, and when?
L2	stage `CONTEXT.md`	What exactly should the agent do in this step?

Each layer **narrows** the context.

L0 stops the agent looking through thirty workspaces when one will do. L1 stops it loading files it will not need until step four — his L1 tables have three columns: Resource, When, and Why. L2 hands it a bounded task with four fields: Input, Process, Output, Completion.

His stated principle:

> *AI agents perform better when they receive narrow, relevant context rather than broad, general context.*

Target size per step: **200 to 400 lines instead of thousands.**

Two more details that matter more than they look:

Naming conventions replace a database. Workspaces are lowercase and hyphenated. Stages are numbered `NN-name`, zero-padded. You do not need an index of what exists, because the names *are* the index.

Explicit exclusion tables. Not just "read this" but "do not read that". Without them an agent wanders, and wandering is what empties your wallet.

Why it landed

Around 30,000 people now work this way. There is a community of over 16,000 members and an academic paper.

It landed because it inverts the usual instinct. The usual instinct, when an AI does not understand your project, is to **give it more**. His answer is to give it **less, but arranged**.

That is the idea this entire book rests on. Everything after this chapter is what happened when someone took it and pointed it at a running commercial system.

If you want the source, go to him. He explains it better than a summary can, and he explains it for the case he designed it for.

Part 3 — What happened when it met a running system

The method was built for workflows: research, then drafting, then review. Numbered stages, an agent moving through them.

What it got pointed at was different. A live product with paying customers — a database, background jobs, payment handling, video generation, email. Nothing in stages. Everything at once, all the time, changing.

Three things happened, in this order.

What carried over unchanged

The routing principle worked immediately, and it worked for the reason he says it does.

The project had a rules file the AI read before every task. 36 kilobytes. Splitting it — a tiny router for *finding*, a separate file for *rules about changing* — cut the cost of a question by a factor of five. Measured, not estimated.

That is his idea, doing exactly what he said it would do, on a system he has never seen.

`CLAUDE.md` as the entry point: his. The small routing table at the front: his. The principle that an agent should read narrowly and stop: his.

You can still see the fingerprints in the real project today. Four files named `CONTEXT.md`, one per area. A folder of numbered work instructions — `1-…`, `10-…`, `14-…`. That is his naming scheme, unchanged.

The first thing that did not fit

ICM routes by **work stage**. A running product has no stages.

`01-discover`, `02-drafting`, `03-review` describes a pipeline — something with a beginning and an end, where step two comes after step one.

Login, billing, video generation, email: these are not steps. They all exist simultaneously. Nobody asks "which stage am I in". They ask "which part of the product do you mean".

So the stage router became a **subject router**. Same principle — one small file, narrow the context, read one thing and stop — but arranged along a different axis.

That was a small change. The next one was not.

The second thing, which changed everything

Here is the fork, and it is worth reading slowly.

ICM organises instructions. What should the agent do, with what, in what order. Instructions are written once and stay correct until you decide to change them.

A running system also needs descriptions of state. How does billing work *right now*? Which tables exist *today*? What does this error actually mean?

And state does something instructions never do: **it goes stale by itself.**

Nobody edits a page to make it wrong. Someone changes the system, and the page — untouched, still confident, still detailed — quietly stops being true. Nothing announces this. There is no error message for a document describing a world that has moved on.

That is the entire problem the rest of this book solves, and it is a problem ICM never had to solve, because instructions do not rot on their own.

What had to be added, and what each one cost to learn

Every addition below came from something going wrong. Part 5 tells each story properly; here is the shape of it.

Addition	The thing that forced it
A date on every page	pages were months out of date and nothing indicated it — not to a human, not to the AI
A machine-readable inventory	the checker needs a list to compare against; you cannot compare prose to reality
Checking in both directions	pages described things that no longer existed. Nothing was missing, so nothing looked wrong
Checking numbers in prose	"27 tables" was written in three places when there were 36
A "Traps" section	hours lost to a silent misconfiguration appear in no source document, anywhere
Enforcement at the save step	an AI followed every single rule and then never saved the work

Look at that list again. Every item is about the same thing: **how do you know the page is still true?**

That is the question a living system forces on you. And it is why this system has a checker that runs as a program, comparing list against list, rather than a periodic review.

This is not a criticism of ICM

It would be easy to read the last section as "his method was incomplete". It was not.

He built a method for organising instructions to agents, and for that it is complete. The additions here are answers to a question his case does not raise. If your work looks like his — pipelines, stages, repeatable workflows — you may need none of this, and adding it would be overhead for nothing.

Take the routing. Add the state-checking only if your ground moves.

A third person, arriving from somewhere else

In April 2026 **Andrej Karpathy** published a pattern called **LLM Wiki**. Different starting point: he was thinking about *sources* — articles, papers, research material — not about agents.

His answer: instead of searching raw sources on every question, have the AI compile them **once** into a linked markdown wiki. Every later question goes to the wiki.

His structure will look familiar by now. An `index.md` catalogue. A `log.md` kept purely chronological. Interlinked pages. A schema file holding the conventions. And a periodic **lint** pass looking for contradictions, superseded claims and orphan pages.

His difference from this book is the same one, seen from the other side: **his raw sources do not change.** A paper published in March says the same thing in December. So a lint that the AI performs by reading and judging is enough.

For a running system it is not, because the ground moves under you — hence a program, not a judgement.

But that comparison cuts both ways, and the honest version is this: the automated check here is only possible because there is something you can *ask*. "Which tables exist right now?" has an answer you can query. In Karpathy's domain there is no such question. He could not build this checker even if he wanted one. That is not an advantage. It is a different kind of problem.

Where that leaves you

Three people, from three directions, with the same underlying conclusion:

> **The expensive part is not thinking. It is searching. So compile once, and afterwards ask only the compilation.**

When an idea arrives three times independently, it stops being a matter of taste.

The rest of this book is the third version of it — the one for a system that will not hold still. Part 4 shows you how it works. Part 5 explains why each piece is shaped the way it is, one accident at a time.

Part 4 — Follow one question through the system

Before any rules, let us just watch it work. One question, start to finish.

Say you ask: **"How does the login work?"**

Without the system

The AI has no idea where anything is. So it does what it can: it reads a lot. Your rules file, several likely-looking files, maybe a folder of old notes. Somewhere in there it finds the answer, and tells you.

You paid for all of it. And because it read old notes on the way, there is a decent chance part of the answer describes how login worked last spring.

With the system

Step one. The AI opens one small file — the *index*. About 4 kilobytes. It looks like this:

```
| Your question is about...      | Read      |
|--------------------------------|-----------|
| Logging in, accounts, security | auth.md   |
| Payments and subscriptions     | billing.md|
| Sending email                  | email.md  |
```

Plus three short rules, which we come to later.

Step two. It finds the matching row. Login → `auth.md`. It opens that one file.

Step three. At the top of `auth.md` is a block called *Quick answer* — three lines covering the three questions people actually ask about login.

If your question is one of those three, **the AI stops here.** Total cost: a small index plus twenty lines.

Step four. If you need more, it keeps reading the same file. The sections are always in the same order: how a request flows through the system, what gets stored, the traps, where the actual files live.

Step five. If your question turns out to touch a neighbouring area, the bottom of the file has a section called *Related* naming the next file by name. One hop, not a search.

That is the whole mechanism.

What just happened

The AI read **one small file plus one medium file.** Not your project. Not your history. Not old notes.

Same answer. Roughly a fifth of the cost. And — this matters more than the money — **it could not accidentally read something outdated**, because it never opened the old material at all.

The one thing to hold on to

Everything else in this book serves that walkthrough. If you remember nothing else:

> **The AI reads a tiny index, then exactly one file, and stops.**

The rest of the system exists to make sure that one file is the *right* one and still *true*.

Part 5 — Why each piece is exactly the way it is

Now the interesting part. Each decision below exists because something failed without it. I give you the rule, then the accident.

Rule 1 — One file per area — never two

The rule: every part of your project gets exactly one file describing how it works *today*.

Why not two? Because two files about the same thing will disagree. Not "might" — will. Someone updates one and not the other, and now you have two sources contradicting each other.

And here is the real damage: **the moment two sources disagree, both become worthless.** You cannot tell which is right, so you have to go and check the actual system — which is exactly the expensive thing the documentation was supposed to save you from.

One file can be wrong. Two files are guaranteed to be untrustworthy.

Why "today"? Because "how it used to work" and "how we would like it to work" are different documents with different purposes. Mixed together, a reader cannot tell which sentence is which. Neither can your AI.

Rule 2 — The index must stay tiny

The rule: the index stays under about fifty lines. Nothing in it except question → file.

Why so strict? Because this file is read *every single time*. Every question, every session, forever.

That makes it the one file where size is a permanent tax. Ten extra lines is not ten lines — it is ten lines multiplied by every question you will ever ask, for as long as the project exists.

The accident: in the real project, the index and the rules were the same file. It grew sensibly, one useful paragraph at a time, until it was 36 kilobytes. Nobody ever added anything unreasonable. It simply accumulated.

That is how you end up paying 10,000 tokens to find out which file to open.

The fix was to split it in two: a tiny file for *finding* things, a separate file for *rules about changing* things. You need the finder every time. You need the rules only when you are actually changing something.

Cost per question dropped by a factor of five. Nothing was deleted. It was moved to where it belonged.

Rule 3 — Every file carries a date

The rule: every page starts with the date it was last verified.

Why: without a date you cannot tell "this is current" from "this has been wrong for eight months". Neither can your AI — it will read a stale page with total confidence and tell you something false with total confidence.

A date turns an unanswerable question ("is this still true?") into a judgement you can actually make ("checked three days ago, probably fine" versus "checked last year, I should verify").

The accident: several files carried dates months older than their last edit. Someone changed the content and left the date alone. The checker now flags any page older than sixty days — not because sixty is magic, but because *something* has to nag.

Rule 4 — Every file has the same shape

The rule: six sections, always the same, always in the same order.

```
Quick answer          three common questions, one line each
How it flows          where it starts, what it touches, where it ends
Data                  what gets stored and what matters
Traps                 things that actually cost someone real time
Where things live     file, function, page, table — the addresses
Related               neighbouring areas, by filename
```

Why identical shape? So a reader can stop early with confidence.

If every file is arranged differently, you have to read all of it to be sure you have not missed the answer. If the shape never changes, you know a quick answer lives at the top and addresses live near the bottom — so you can stop the moment you have what you need.

Predictability is what makes early stopping safe. And early stopping is where the daily savings actually come from.

Rule 5 — The "Quick answer" at the top

Why it exists: most visits to a file are not deep research. They are "wait, which one was it again?"

Three lines at the top answers those, and the reader — human or AI — closes the file after twenty lines.

Here is a real one:

```
- Sacred rule: leads are never discarded. Over the limit they are
  still captured and only LOCKED.
- Where does the lock take effect? In the database permission rule,
  not in the interface — that is why it cannot be bypassed.
- Which endpoints are public? Three, deliberately — which is why
  rate limiting is still an open item.
```

Someone who reads only that already knows more than most people who read the whole file.

Rule 6 — The "Traps" section — where the real value hides

The rule: write down things that actually cost someone hours. Not theory, not best practice. Scars.

Why this is the most valuable section in the whole system: everything else can, in principle, be re-derived. Someone can read the code and work out what it does.

Nobody can re-derive the three hours you lost. That knowledge exists in exactly one place — your memory — and it evaporates.

Three real ones:

> **A file-exclusion list does not accept comments at the end of a line.** Writing `folder/ # too big` creates a rule for a folder literally named `folder/ # too big`. It excludes nothing, and it fails silently. The first attempt backed up 470 megabytes instead of 8.
> **That same exclusion list only applies to files not already tracked.** Anything already tracked stays tracked, no matter what the rule says. It has to be removed explicitly, by hand.
> **One error code meant "the gateway gave up waiting", not "the work failed".** The work had actually finished successfully. Days went into fixing the wrong end of that.

None of those are in any manual. That is exactly why they belong in yours.

Rule 7 — "Related" at the bottom — what turns a pile into a map

The rule: every file ends by naming its neighbours.

Why: without it you have twenty files and no idea which one you need. With it, every file is a junction — you land anywhere and can walk to the right place.

The practical effect: **from any question, at most two hops.** One to the index, one to a neighbour if you guessed slightly wrong.

Leave this out and the system still technically works, but people stop trusting it — because guessing wrong costs them a search, and after two bad searches they go back to reading everything.

Rule 8 — The history is separate, and forbidden for learning

The rule: you keep a diary of what changed and why. You never read it to learn how something works.

Why keep it at all? Because it stops you re-litigating decisions. "Why don't we do X?" — because we tried X in March, and here is what happened. That is what stops a project going in circles.

Why forbid it for learning? Two reasons.

It grows without limit. The real project's diary is now 93 files. Reading it to understand the system means reading years of change notes to reconstruct a present state that is written down plainly two folders away.

It is full of things that stopped being true. An entry from March describes March. It was accurate when written and is misleading now. Nothing marks it as expired, because it was never meant to be read as current.

Think of a car. The diary is the service history: "replaced the brakes in March, they squeaked." The manual explains how the brakes work. Reading five years of service history to understand brakes would be slow, confusing, and leave you believing things that stopped being true in 2019.

This rule is written in plain words in the first file the AI reads. Not buried in a style guide — right at the front, where it cannot be missed.

Rule 9 — A machine-readable inventory

The rule: alongside the human pages, keep one file listing what actually exists in the running system — every database table, every endpoint, every scheduled job — in a format a program can read.

Why: because the checker needs something to compare against.

You cannot compare prose to reality automatically. You *can* compare a list to a list. The inventory is the bridge between "what is really running" and "what our pages claim".

It is a snapshot, not a live feed. So it carries its own refresh instructions: the actual commands that answer "what exists right now?" in your particular setup. Refreshing it takes two minutes, and everything downstream depends on it being honest.

Rule 10 — The checker runs in *both* directions

This is the piece most people never build, and the one that matters most.

Direction one — what exists but is documented nowhere. Obvious and useful. Someone builds a new thing and forgets to write it down; the checker notices something in the inventory that appears on no page.

Direction two — what is documented but no longer exists. Not obvious. And this is where documentation actually rots.

Think about it. Documentation rarely dies by going missing — you notice missing. It dies by **describing a world that has moved on.** The page is still there, still confident, still detailed, and quietly wrong. Nobody notices, because nothing is absent.

The accident: the first version of this check produced false alarms. It flagged anything hyphenated — storage names, ID codes, style names — as a possible dead reference. Narrowing it to lines actually talking about endpoints fixed it. Worth knowing: **a check that cries wolf gets switched off,** and then you have no check at all.

A later addition: numbers in prose. The pages said "27 tables" in three separate places when there were 36. No structural check could see that — it was just words. Now the checker compares any number over twenty against the inventory and complains when they disagree.

Rule 11 — What the checker deliberately cannot do

It cannot tell whether you are right.

It verifies that a page exists, has a date, has its sections, is reachable, and does not describe things that are gone. It has no opinion on whether your explanation of how login works is correct.

Green means "nothing is structurally missing". It does not mean "this is true".

I labour this because the failure mode is nasty: a green checker *feels* like approval. If you start believing it validates content, you stop verifying facts against the real system — and the whole thing becomes a very tidy way of being confidently wrong.

The honest position is printed in the checker's own output, every single run: *this only checks structure. Verify the facts against the running system.*

Rule 12 — Three copies

The rule: the copy you work on, one online, one somewhere else again.

Why three and not two? Two copies protect you from one failure — your laptop dies, the online copy survives. But if those two are your only copies, one bad sync can propagate a mistake to both.

The third is usually a different *kind* of place — a server, a different provider — which means it fails for different reasons.

The accident: in the real project this was not done at all. Everything existed exactly once, on one laptop. No history, no backup, no way back. Months of work, one hard drive.

That was the single biggest risk in the entire system, and it had nothing to do with documentation quality. It was found by accident, while looking at something else.

Rule 13 — The entry point lives *inside* the project

The rule: the file that tells an AI where to start belongs in the project itself, not in a folder next to it.

Why: a file outside the project is invisible to anyone who is not you, on your machine, today.

The accident — and this one is mine. I put the entry point in a folder on the user's own computer, outside the project. It worked perfectly. It was also not backed up, would not survive a lost laptop, and would not travel with the project if it were ever sold or handed over.

Worse: a *different AI* was working directly inside the live project, where that file did not exist. It could not possibly follow rules it could not see. I had built a system that only worked for me, on one machine, and I had not noticed.

The entry point belongs wherever the work happens. If work happens in two places, it goes in both.

Rule 14 — Every rule states its reason

The rule: never write "do X". Write "do X, because Y".

Why: because the reader is an AI weighing instructions against a situation nobody anticipated. An instruction with a reason survives that. One without a reason gets dropped the moment it looks inconvenient.

"Never load the history folder" invites an exception the first time something seems missing.

"Never load the history folder, because it is 93 files of outdated change notes and you will burn a fortune to learn things that stopped being true" does not.

Reasons are not decoration. They are the enforcement mechanism for everything a program cannot check.

Rule 15 — Rules and enforcement are different things

The rule: wire the checker into a step that already has to happen — saving, or deploying. Never as a separate thing to remember.

The accident, and it is the best story in this book: a different AI built an entire new feature in the real project. It read the entry point, found the rules, and followed every one of them. It wrote the documentation, updated the index, refreshed the inventory, wrote the diary entry. Genuinely good work, unprompted.

It never ran the save command.

An hour of excellent work sat unsaved on one laptop — not backed up, not online, one accident away from gone.

The rules worked. The safety net was switched off.

That is the distinction. Rules protect you from forgetting. Enforcement protects you from the day someone skips the rules — and "someone" includes every AI you will ever use.

Part 6 — What happens when your project grows

Fair question at this point: does this survive contact with a growing project? Or is it a tidy structure that falls apart the moment real work happens?

It got tested by accident, and the test is worth walking through.

The setup

One AI (Claude) built the documentation system in the morning. That afternoon a **completely different AI** — Codex, made by a different company — was asked to build a new feature. A big one: nine new database tables, a new interface, three new database functions.

Nobody mentioned documentation. Nobody said "and don't forget the rules". The two AIs never communicated.

What happened

Codex opened the project, found the entry file, and read it. The rules were in there, each with its reason.

Then it built the feature — **and wrote the documentation as part of the job.** It updated the page for that area, rewrote a warning that was no longer true, added a row to the index, refreshed the inventory, and wrote a diary entry explaining why the feature was designed the way it was.

The checker went from 74 documented items to 87. All green.

What it missed

Four things, all small, all in places the checker cannot see:

1. **A cross-reference.** The page did not link to the new diary entry, so a reader could find the *what* but not the *why*.
2. **Open tasks in the wrong place.** Four things still to do were written in the diary instead of the to-do list. Anyone looking for "what is still open" would not have found them.
3. **A status line** on the overview page still said the old thing.
4. **Numbers in prose.** Three places still said 27 tables. It was 36.

Twenty minutes to find and fix. Number four is now caught automatically — that check exists *because* of this.

Why this is a good result

Look at the alternative.

Without any of this, that feature would have been built and documented nowhere. Nine tables, a new interface, three functions — discovered three months later by someone who has to reverse-engineer it from the code, wondering whether it is even still in use.

Instead: a different AI, from a different company, with no briefing, produced correct documentation as a side effect of doing its job.

That is the actual promise. Not perfection. Continuity — the next one picks up where the last one stopped, and you explain nothing.

And a fifth miss, which matters more

Codex never ran the save command. See rule 15 in Part 5. The work sat unsaved.

I include this because a book that tells you only the good half is selling you something. The system held. The safety net was not switched on. Both are true, and the second is fixable in about a minute.

Part 7 — Setting it up by hand

You can skip this part entirely.

The appendix has a prompt. Paste it into your AI, answer three questions, and everything in this book gets built for you — including the checker, tested.

This part exists for two reasons. Some people want to do it themselves. And more importantly: **so that nothing in the system is a black box.** Even if you never type any of it, reading this means the folder your AI creates will make sense when you open it.

What gets created

```
your-project/
├── CLAUDE.md              entry point for Claude
├── AGENTS.md              entry point for other AIs + the rules
├── ROUTING.md             question -> file  (the tiny index)
├── START-HERE.txt         the same thing, for humans
├── .gitignore             what not to save
├── references/
│   ├── _TEMPLATE.md       the required shape of a page
│   └── ...                one file per area — grows over time
├── history/
│   └── README.md          "this is history, never load it to learn"
├── system-inventory.json  what actually exists, machine-readable
├── EMERGENCY.md           where everything lives, who owns what
└── ops/
    ├── check.*            the checker
    └── publish.*          check, save, copy everywhere
```

On day one, `references/` is empty and `ROUTING.md` has no rows. **That is correct.** The skeleton is valid when empty. Areas appear as you build them.

The six steps

1. Take stock — and do not guess. What parts does your project actually have *today*? Not plans. Five to twenty is normal.

> The most expensive mistake I made while building this was concluding things from a summary instead of going and looking. Twice. Both times I was wrong, and both times I had to be corrected. Look at the real thing.

2. One page per area, using the six sections from rule 4 in Part 5.

3. The index, using the shape from rule 2 in Part 5. Under fifty lines. If your project has a word that means two different things, warn about it right at the top — in the real project "avatar" meant both a customer persona and a talking presenter in a video, and that one warning has saved hours.

4. Version control and copies.

```
git init
git add -A
git commit -m "First version of the documentation"
```

Creates a history and saves the current state. From here you can see what changed and go back.

Then a **private** repository online:

```
git remote add origin https://github.com/YOURNAME/YOURPROJECT.git
git push -u origin master
```

Tells your folder where its online copy lives, and uploads it.

Private, not public — these pages describe how your system is built and what is not yet secure about it.

Never write a password or key into any of these files. The name of a setting, never the value. This documentation will end up on other machines.

5. The checker, implementing rules 10 and 11.

6. Break it on purpose. Put a wrong number in, run the checker, confirm it complains, put it back.

> An untested check is worse than no check, because you will trust it exactly when you should not. Everything else in this list is negotiable. This step is not.

Making it unskippable

Wire the checker into saving or deploying — see rule 15. Not as a separate command anyone can forget, including your AI.

Part 8 — Connecting your AI

Written 2026-07-26. This chapter ages fastest, because these conventions change. If something here does not match what you see, trust the tool's current documentation.

How it works

Most AI coding tools look for an instruction file when they open a project. Put your entry point there and you never explain anything again.

Two filenames cover most tools today:

- `CLAUDE.md` — read by Claude Code
- `AGENTS.md` — a vendor-neutral convention, read by Codex and others

Write both. They are short and can say nearly the same thing. Two files cost you nothing. Having none means all your work is invisible.

What goes inside

Short. This is read every session, so every line is a permanent cost — the exact mistake from rule 2.

```
# Start here

Read ROUTING.md first. It tells you which single file answers the
question. Then read only that file.

Three rules before you load anything:

1. The history folder is history, not the current state. Never load
   it to learn how something works — it is large and full of things
   that stopped being true.
2. If you are CHANGING something, also read AGENTS.md.
3. <your project's one confusing word, and what it means where>

After changing anything: update that area's page in the same task,
set its date to today, then run the checker.
```

Resist explaining your project here. That instinct is what produced the 10,000-token file.

Checking that it actually works

Do not assume — test it.

Start a completely fresh session and ask something you know is documented. If the answer comes back without you explaining anything, it works. If the AI asks "what is this project?", it never found your file.

For a tool that reads nothing automatically, paste this first:

```
Project: <name>, at <path>.

Read ROUTING.md first — a short file telling you which ONE file
answers my question. Then read only that file.

Do not load the history folder to learn how things work.

If you change anything: update that area's page in the same task and
run the checker before finishing.

My task today: <task>
```

What to expect

Not perfection. See Part 6: a different AI followed every rule unprompted, missed four small things, and skipped the save step.

That is the realistic outcome, and it is far better than the alternative.

What you actually get

Cheaper. A small index plus one page, instead of everything you own. About five times less, measured.

Not locked in. Claude, Codex, whatever comes next — they all read plain text. When something better arrives you switch and lose nothing. Your knowledge is not stored inside anyone's product.

No more re-explaining. New session, new tool, new day. It reads the entry point and knows where it is.

Something you could hand over. Sell the project, take on a partner, or come back after a year away — it explains itself. That last one will happen to you for certain.

The seven things worth remembering

1. **Nine out of ten tokens are usually navigation, not answers.** Measure before you assume.
2. **One page per area, describing today.** Two pages about one thing make both untrustworthy.
3. **The diary is not the manual.** Never learn how something works from a change log.
4. **Write down what cost you time.** Everything else can be re-derived. That cannot.
5. **Check in both directions.** Documentation rots by describing what is gone, not by going missing.
6. **Test the checker by breaking something.** An untested safety net is worse than none.
7. **Rules are not enforcement.** An AI followed every rule and still skipped the save.

Appendix — Ready-made prompts

Copy these. Replace anything in `<angle brackets>`.

A. A brand new project — build the skeleton

For day one, when **nothing** exists yet. It does not write content; it builds the empty skeleton and installs the rule that makes it grow by itself afterwards.

```
I am starting a new project. Nothing exists yet — no code, no database,
nothing. Before we build anything, set up the skeleton that lets any AI
know instantly what this is, without me explaining it.

FIRST ask me these three things and wait for my answer:
1. What is the project called, and what should it do in one sentence?
2. What is it built with — language, database, hosting? If something is
   undecided I will say "open". Then leave that spot marked TO BE FILLED
   IN rather than guessing.
3. What operating system am I on? That decides whether the scripts are
   .sh or .ps1.

Then create exactly this structure — empty, but complete:

  ROUTING.md           Question -> file. An EMPTY table at first, plus
                       the three loading rules. Keep under 50 lines.
  AGENTS.md            the working rules (below). Read only when
                       CHANGING something, not when asking.
  CLAUDE.md            short entry point, points at ROUTING.md.
  START-HERE.txt       the same thing for humans, with copy-paste
                       snippets.
  references/_TEMPLATE.md  the required shape of an area file.
  history/README.md    states plainly: this is history and is NEVER
                       loaded to learn how something works.
  system-inventory.json  empty lists plus a _refresh block containing
                       the actual commands that answer, in THIS stack,
                       "what exists right now?"
  EMERGENCY.md         where everything lives, who owns what, how to
                       come back after losing this machine. Notes at
                       first — it grows.
  ops/check.*          the checker (below).
  ops/publish.*        checks, commits, pushes to every copy.
  .gitignore           comments on their OWN lines only. A comment at
                       the end of a line becomes part of the rule and
                       silently makes it do nothing.

AGENTS.md must contain these rules, each WITH its reason — a rule
without a reason gets ignored by the next AI:

  1. Exactly ONE reference per area, describing how it works today.
```

```
     Why: two files about the same thing disagree within a month, and
     then nobody trusts either.
  2. When a new area appears, its reference and its ROUTING.md row
     appear in the SAME task. Why: "later" means never.
  3. When something changes, its reference is updated in the same task
     and its date set to today.
  4. History is history, not the current state. Why: it grows without
     limit and contains things that stopped being true. Learning from
     it means learning wrong things and paying for the privilege.
  5. Never write a secret value into a file — only the NAMES of
     settings. Why: documentation leaves the machine.
  6. Anything you did not verify, mark as unverified. Do not guess and
     present it as fact.
  7. Before saying "done": run ops/check, then ops/publish.

The checker must fail when:
  - a reference has no date, or the date is older than 60 days
  - a required section is missing
  - a reference is unreachable from ROUTING.md
  - an internal link points at nothing
  - something in system-inventory.json appears on no page
  - a page describes something no longer in the inventory
  - a number in the prose no longer matches the inventory

With ZERO areas it must pass green. The skeleton is valid even when
nothing is in it yet.

Finally, and this is the most important part:

  a) Run the checker and show me it is green.
  b) Break something on purpose — add a reference with no date — run it
     again, show me it complains, then undo it. Do not tell me it works
     without demonstrating it.
  c) git init and a first commit. Then tell me the two commands I need
     to create a copy outside this machine.

And from now on, for everything we build together: every new area gets
its reference and its routing row in the same task. I should not have to
remind you.
```

The last paragraph is the actual trick. Everything above it is one-time setup. That sentence is what makes the system grow without you — and it lives in `AGENTS.md`, so tomorrow's AI reads it too.

At the start `ROUTING.md` has zero rows and the checker has nothing to check. That is correct. The first area appears the moment you build the first piece — and from then on the rule carries it.

B. Do it for a project that already exists

```
This project already exists and is running. I want documentation that
reflects REALITY, not what anyone intended.

Start by measuring, not writing:

1. Inspect the live system directly. List everything you find and say
   how you found it. For anything you could not check, say so plainly.

2. Compare that against any documentation that already exists. Report
   three lists separately:
   - things that exist but are documented nowhere
   - things documented that no longer exist
   - things documented differently from how they actually are

3. Show me those lists BEFORE writing anything. Some gaps will be
   deliberate and I need to tell you which.

Then build the same skeleton as in prompt A (ROUTING.md, AGENTS.md,
CLAUDE.md, references/, history/, system-inventory.json, ops/check,
ops/publish) — except this time you fill it immediately with what you
found in step 1, instead of leaving it empty. Same rules, same checker,
same proof at the end: break it on purpose and show me it complains.

Two things I especially want to know:
- Anything that looks dead but might not be. Check whether something
  actually uses it before you call it dead.
- Anything that belongs to a different project or owner than its name
  suggests.
```

Both of those last two points come from real findings. In my own project a service named after the project turned out to belong to a *different* business entirely, and switching it off would have broken something live. Names lie. Check usage.

C. Keep it healthy — say this when starting work

```
Read TABLE-OF-CONTENTS.md, then only the file it names for my task.
Do not load the history folder.

When you change anything: update that area's page in the SAME task,
set its date to today, add a history entry, and run the checker before
you tell me you are done.

My task: <task>
```

D. The monthly check-up

```
Run the checker and report what it says.

Then check three things it cannot see:
1. Any numbers written in the documentation that no longer match
   reality (counts of tables, endpoints, prices, limits).
2. Any area where the live system has changed but the page has not —
   compare dates against recent history entries.
3. Any page whose "Related" section points at something that has
   since been renamed or merged.

Report findings. Do not fix anything until I have seen the list.
```

One last thing

The hardest part of this is not the setup. It is the first honest look.

When I did this for a real project, the inspection turned up things nobody wanted to find: a service that had been running for months with nobody sure what it did, two separate businesses sharing one machine so neither could be sold without the other, an entry point that would have vanished with a dead laptop, and 470 megabytes of junk about to be saved forever because of a misplaced comment.

None of that was in anyone's documentation, because nobody had ever gone and looked.

Go and look. Write down what you find. Then let the AI keep it that way.

Written using the system it describes. Every number was measured on a real project. Every mistake actually happened.