A TERRABYTE FIELD GUIDE

The AI Master Guide

TERRABYTE SOLUTIONS

> By the end of this guide you'll run any AI project out of a folder of plain-English files — and it'll pick up right where you left off, every time. No coding. If you can write an email, you can do this.

Welcome

Most people open Claude, type a question, and start over every single time. Half their words go to re-explaining who they are and what they're working on. Then they hit the limit, open a new chat, and do it all again tomorrow.

This guide fixes that. But it's really about something bigger, and I want to say it plainly up front because it's the thing that changed how I work:

In 2026, the most valuable language you can learn is English.

Not Python. Not a new app. The skill that compounds is writing plain instructions, in plain text, organized so an AI can find them. Andrej Karpathy — who used to run AI at Tesla — put it best back in 2023: *"The hottest new programming language is English."* Three years later that's not a clever line anymore. It's just how the work gets done.

What You’re Actually Going to Do

So you know what you’re building toward, here it is in one sentence:

You’re going to build a folder, and that folder is going to make your AI noticeably better.

Not a better prompt — a better **input**. Right now what you hand the AI is a paragraph you typed from memory. By the end of this guide, what you hand it is a small, deliberate structure: who it is, what the project is, where everything lives, what to read and what to ignore. Same model, same subscription, same twenty bucks a month — dramatically better output, because you changed what goes in.

That’s the trade the whole guide is about. **You stop working on your prompts and start working on your inputs.** People will ask what model you’re using. That was never it.

Read this straight through once — about half an hour — to get the shape of the system in your head. Then come back with a real project and follow along with your hands on the keyboard.

Let’s go.

PART 1 OF 9

The Beginning

Two things to set up. Do it once. You never redo it.

Two Things You Need

1) An AI subscription — and its agent harness

These come as a pair, so buy them as one decision.

The **subscription** is the model. The **agent harness** is what lets it out of the chat box — it runs on your computer, opens your folders, reads your files, and writes new ones. That's the whole difference, and it's a big one: a chat window can *talk about* your project; a harness can *work in* it.

Every major provider now ships one, and you use the one that matches the subscription you're already paying for:

Your subscription	Your agent harness
Claude	**Claude Code**
ChatGPT	**Codex**
Gemini	**Antigravity** (not Gemini CLI)
Grok	**Grok Build**

Pick one lane and stay in it. Everything here carries over to all of them — same folders, same files, same ideas. What changes is the command you type and one filename: each harness auto-loads its own map file (Claude Code reads `CLAUDE.md`; some others read `AGENTS.md`). Check yours once, up front. **I use Claude, so every example here says `CLAUDE.md`.**

Roughly **$20/month** gets you started: Claude Pro is $20 and its pricing page says *"Includes Claude Code"* — that's the setup this guide assumes. Claude Code needs a paid plan; the free tier doesn't include it, and free tiers hit their daily limits about ten minutes into real work anyway.

Installing Claude Code

You install it by typing one line into a **terminal** — the plain text window your computer already has, where you type commands instead of clicking. Open it first:

- **Windows** — press the Windows key, type `powershell`, press Enter.
- **Mac** — press Cmd+Space, type `terminal`, press Enter.

A window opens with a blinking cursor. Paste in the line for your system — **the command only** — and press Enter. Copy it exactly; the dashes and slashes are load-bearing.

Windows:

```
irm https://claude.ai/install.ps1 | iex
```

Mac / Linux:

```
curl -fsSL https://claude.ai/install.sh | bash
```

Then type `claude --version`. A version number means you're done — the exact digits don't matter.

If you get `'claude' is not recognized` or `command not found` instead, nothing is broken. The install worked; a terminal window that was already open just doesn't know about it yet. Close that window completely, open a fresh one, and try again.

Then there are exactly **two ways to work**, and you only need one:

- **The terminal** — point it at your project folder, then type `claude` and hit enter. *(How to point it: Part 3, Step 2.)*
- **The desktop app** — if you'd rather click than type. Open it, click the **Code** tab, set Environment to **Local**, and hit **Select folder**. *(On Windows, install Git for Windows first and restart the app — the Code tab needs it.)*

Same engine either way. Pick whichever feels less scary today; you can switch any time.

> **Your first launch.** The first time you run `claude`, it asks a few one-time questions. This is setup, not an error.
>
> - **Pick a theme.** Any of them — it's just colors.
> - **Log in with your Claude account.** It opens your browser; sign in with the account you subscribed with. *(Subscribing and being signed in are two different things — this is the step that connects them.)*
> - **"Do you trust the files in this folder?" → yes.** It's your folder. You'll get this once per new folder, so expect it again in Part 3.
>
> Then you get a `>` prompt. Type in plain English, exactly like the chat box. `/exit` when you're done.

2) Obsidian — your text editor

You need something to write and read `.md` files. Most guides send you to VS Code here. But you're not writing code — **you're writing markdown** — and there's an app built for exactly that: **Obsidian**.

Three reasons it's the right pick:

- **It's a markdown editor, not a code editor.** Its default mode is called *Live Preview*: type `**bold**` and you watch it turn bold. Your headings look like headings, your tables look like tables. You *see* the structure you're building while you build it — which, as Part 2 will show you, is the entire point.
- **A vault is just a folder.** Obsidian calls what it opens a *vault*, but there's no database and no lock-in underneath — in their own words, "a vault is a folder on your local file system." It's your `.md` files sitting on your disk, readable by anything. That matters here, because **your Obsidian vault and your AI's workspace folder are the same folder.** One folder, two windows on it: you in Obsidian, the agent in its own window. Obsidian refreshes automatically when something else edits a file, so you literally watch your agent's work appear.

- **It’s worth learning anyway.** This isn’t a fringe pick — Obsidian’s own CEO publishes a set of agent skills that teach coding agents (Claude Code by name) how to drive a vault. The whole app is built on local plain text, which is exactly what makes it good for you *and* good for an agent.

It’s free — no sign-up, and free for work use too. Download and install it at `obsidian.md`. That’s all for now; you’ll point it at your workspace folder in Part 3, once that folder exists.

> **That’s the whole setup.** You don’t need VS Code, Cursor, an IDE plugin, or an extension to do anything in this guide. Two windows on one folder — that’s it. (Obsidian is an editor, not a terminal, so you still launch your agent the other way: a terminal window, or the desktop app.) And this is the only place in the guide you install anything: from here on, every upgrade you make is just editing text files.

PART 2 OF 9

How to Talk to Your AI

A quick diagnostic for when the output is bad — and then the one piece of syntax that carries the rest of the guide.

The 5-Part Prompt Framework — Skim This

Straight with you: **I don't sit down and build prompts this way, and you won't either.** What follows is a mental model, not a ritual. It's the checklist you run *when something isn't working*, to find the piece you left out. Read it once, know where it lives, and move on — the real leverage is Part 3.

Five parts. Most asks use two or three of them.

Part	The question it answers	Example
Identity	Who is the AI right now?	*"You are a senior B2B SaaS copywriter."*
Task	What, exactly, needs doing?	*"Write a 200-word description of our onboarding feature."*
Context	What does it need to know?	*"Audience is mid-market HR directors. We lost the last two deals on price."*
Constraints	What should it avoid?	*"No jargon. Under 300 words. Don't open with 'In today's world.'"*
Output format	What should the result look like?	*"Three options, each with a one-line rationale."*

Two of the five do most of the work, and they're the two people skip:

- **Context is king.** When a result feels generic, the fix is almost always *more context* — not a cleverer prompt. Guessing is where AI output goes sideways.

- **Every constraint you set is a mistake it won't make.** Every recurring annoyance you have with AI output is an unset constraint. Say it once, out loud, and it stops happening.

And when a job is bigger than one prompt, **split it.** *"Write me a full marketing strategy with a content calendar, an email sequence, and social posts"* gets you something shallow across the board. Outline the themes → you pick one → draft the calendar → you review → write the first email. You correct between steps, and when something's wrong you redo one step instead of the whole project.

The Part That Actually Matters: Markdown

Now the piece I *do* use every single day.

Stop for a second and think about how you read.

A wall of gray text versus one **bolded phrase** in the middle of it. A heading that tells you what's coming versus a paragraph you have to decode to find out. A numbered list versus *"first, and then, after that, and also."* ALL CAPS IN A TEXT MESSAGE versus lowercase. A word in *italics* — you probably heard it emphasized in your head just now.

None of that changed the words. It changed what you noticed, how fast you found it, and how sure you were that you got the point.

That's context you're transmitting without saying it. You've been reading it fluently your whole life and you almost never think about it.

Here's the thing: **the AI reads it too.** Markdown is just the agreed-upon way to write those signals down in plain text so they survive the trip. A heading says *"new topic starts here."* A bullet says *"these items are peers."* A table says *"these things vary along the same axes."* A code block says *"this is literal — don't interpret it."* You're not decorating. You're labeling structure — and structure is context.

That's why every file in this guide is markdown, and it's why the whole thing works. You already know most of it from texting, Reddit, and Slack:

You type	You get
`# Title`, `## Part`	headings (more `#` = smaller)
`**bold**`	**bold**
`*italic*`	*italic*
`- item`, `1. item`	bulleted & numbered lists
`[label](url)`	a clickable link
`> note`	a quoted callout
`` `code` ``	inline `code` — literal text

There's more when you want it — tables, images, checklists, fenced code blocks (three backticks). Don't memorize any of it. The point is that this syntax is light enough for *you* to read and write, yet structured enough for an *AI* to parse exactly. That dual nature — effortless for a human, unambiguous for a machine — is the whole reason markdown, and not Word or a database, became the language everything here is written in.

(This is also where Obsidian earns its keep: type `**bold**` and you watch it become bold. You see the structure land.)

Two of Those Five Have a Home

Look back at the five parts. Two of them — **Identity** and **Context** — barely change from one ask to the next. Retyping them every session *is* the waste this whole guide is out to kill.

So we stop. Those two get a permanent home in plain markdown files that the AI reads on its own; the other three you just say, fresh, each time. That's the entire leap of the next part — and it's why this small framework is really the whole system, shrunk down to one message.

PART 2 RECAP

- The 5-part framework is a *diagnostic*, not a ritual. Run it when output feels generic.
- Context is king; every constraint you set is a mistake it won't make.
- Chunk big jobs into one-clear-ask steps.
- **Markdown is structure, and structure is context** — the same signals you already read as a human, written down so the AI gets them too.
- Identity and Context barely change between asks. Next, we give them a home in files.

PART 3 OF 9

Your First Workspace

Where the upgrade actually happens. One folder, a few honest files — and an AI that already knows the story.

Two Words You'll Use for the Rest of This Guide

Before we build anything, name the two things:

- **Workspace root** — *the folder you launch your agent in.* Often just "your workspace." It's the top of the world as far as the AI is concerned: everything it can reach lives inside it, and it starts reading from the top.
- **Rooms** — *the folders inside your workspace,* each one doing a single job.

A workspace, with rooms in it. That's the whole vocabulary, and it's worth using precisely, because in a minute the difference between "a folder with stuff in it" and "a workspace with rooms" is going to be the difference between a decent answer and a great one.

Stop Chatting. Start Building.

Here's the move that quietly separates the people who get a lot out of these tools from the people who don't: **stop chatting, and start building a workspace the AI reads every time.** We'll build a real one right now — a little workspace that turns your experience into job applications. Follow along; you'll have something useful by the end of this part.

Step 1 — One Folder, a Few Honest Files

Create a folder on your Desktop and call it `my-career`. That's your workspace root. Now open Obsidian, choose **Open folder as vault**, and pick `my-career` — leave that window open; it's the second of the two windows from Part 1.

Inside it, make three files with **New note**, named `CLAUDE`, `wins`, and `CONTEXT`. Obsidian saves each one as a `.md` file for you — markdown, the syntax from Part 2.

> **Make them in Obsidian, not in File Explorer or Finder.** Both hide file extensions by default, so a new text file you name `CLAUDE.md` silently becomes `CLAUDE.md.txt`. It looks right, and your agent will never read it. Obsidian sidesteps the whole trap.

Use your own details as we go; mine are just here to make it concrete.

`CLAUDE.md` — who the AI is, and the rules

This is the one filename that's special: when you start your agent in a folder, it reads `CLAUDE.md` **automatically**. So it's where the AI learns who it's working for.

```
# Identity

You help me turn my real experience into resumes, cover letters,
and LinkedIn posts. I'm Sam, aiming for junior data-analyst roles.

## Rules
- Only use what's in this folder. Never invent jobs, numbers, or skills.
- Plain, confident language — no buzzwords like "synergy" or "rockstar."
- If you're missing something, ask me. Don't guess.

## Where things live
- `wins.md` — the honest list of what I've actually done. Start here.
- `CONTEXT.md` — what I'm focused on right now.
```

That "never invent" rule is small and mighty — it's the difference between a resume you can stand behind in an interview and one that trips you up. And that last bit — *Where things live* — isn't decoration: it's the only way the AI knows those other files exist at all. Hold that thought; Step 2 makes it explicit.

`wins.md` — the real raw material

The honest list of things you've actually done. Don't polish it; just get it down.

```
# What I've actually done

- Cut a weekly sales report from 6 hours to 20 minutes with a pivot
  table and a few formulas. (Retail job, 2024-2025.)
- Taught myself SQL and built the queries behind our team's inventory
  dashboard — 8 people use it daily.
- Capstone: analyzed 3 years of campus-dining data, found the two
  dishes driving 40% of food waste, presented it to facilities.
```

Tip — point it at what you've already got. You don't have to type your history from memory. Just say: ***"Go look through [my old resume / my projects folder / my portfolio] and pull my real wins into*** `wins.md`**,"** swapping in whatever you have (a folder of past work, a LinkedIn export, an old CV). The agent opens it and drafts the file for you. *(Files on your computer work right now; to let it reach your GitHub or Google Drive directly, that's Part 6.)*

`CONTEXT.md` — what you're doing right now

```
# Right now

Targeting junior data-analyst roles at small companies.
Applying to ~3 a week — each needs a resume and a short cover letter.
```

This is the file that changes most often. Keep it current; it's how the AI knows what *today* is about.

Step 2 — Point Your Agent at It

Two ways in, and they do the same thing:

- **Terminal:** point it at the folder first. In File Explorer, open `my-career`, right-click any empty space inside it, and choose **Open in Terminal**. *(Mac: open Terminal, type* `cd` *— with the space — then drag the* `my-career` *folder from Finder into the window and hit enter.)* The folder name now sits next to your cursor; that's how you know you're in the right place. Now type `claude`.

- **Desktop app:** click the **Code** tab, set Environment to **Local**, and **Select folder** → `my-career`.

Either way, it reads your `CLAUDE.md` automatically the moment it starts. *(Type `/context` to see exactly which files actually loaded — a useful habit. If `CLAUDE.md` isn't in that list, you either launched in the wrong folder or you've got a `CLAUDE.md.txt`.)*

Now ask for something real: *"Using my wins, draft a three-line LinkedIn 'About' for a junior data analyst."* Watch what happens — it writes in your voice, from your actual experience, no buzzwords, because you set the rules once and it can see the material. That's the whole difference between *prompting* and *directing*.

The first time it wants to create or edit a file, it stops and asks. Say yes. **That pause *is* the harness:** a chat window can only talk about your files; this one is asking to touch them.

> **The one rule that explains everything: context is king — and `CLAUDE.md` is how you crown it.** The AI only *reads* a file if something it already has points it there. `CLAUDE.md` loads automatically, every time — but every *other* file has to be **reachable from it**. That's exactly why we listed `wins.md` and `CONTEXT.md` *inside* `CLAUDE.md`: not as decoration, but as a chain of references the AI can follow. The output is only ever as good as what's reachable — so the real skill isn't a cleverer prompt, it's controlling what the AI can see.
>
> **This is already the foundation.** If you only ever did this much — a folder, a few honest files, an AI pointed at them — you'd be ahead of almost everyone else using these tools.

Step 3 — The Move That Sets You Apart: Rooms

One folder with files is great. The leap most people never make is to **split the workspace into rooms** — each for one job — with a map up top that sends the AI to the right one.

Watch `my-career` grow:

```
my-career/                ← the workspace root: you launch the agent here
├── CLAUDE.md             ← the map: who I am + where everything lives
│
├── 01-my-work/           ← source of truth: everything I've actually done
│   └── CONTEXT.md
├── 02-a-job-appears/     ← paste a posting → research it, pick what to feature
│   └── CONTEXT.md
└── 03-applications/      ← the finished, tailored resume + cover letter
    └── CONTEXT.md
```

Your `wins.md` moves into `01-my-work/` — it's the source of truth now — and your root `CONTEXT.md` folds up into the map itself: *what I'm focused on right now* becomes a line at the top of `CLAUDE.md`, and from here on `CONTEXT.md` means a **room's** file. That little *Where things live* list grows up too: the top `CLAUDE.md` becomes a full **floor plan** — a table of contents with a routing table, now pointing at *rooms* instead of single files.

```
# My Career Workspace

I'm Sam, aiming for junior data-analyst roles.

## Rooms
| When I…                    | Go to             | Read first |
| -------------------------- | ----------------- | ---------- |
| Add or update what I've done | 01-my-work/       | CONTEXT.md |
| Work a specific job posting  | 02-a-job-appears/ | CONTEXT.md |
| Build the resume / letter    | 03-applications/  | CONTEXT.md |

## The one rule
Everything traces back to 01-my-work. Never invent — pull from there.
```

Notice the `CONTEXT.md` **in every room.** Here's the idea, and it's worth getting right:

- Only the map file is read *automatically*. It's the front door to the workspace. In Claude Code that file is `CLAUDE.md` — note that it reads `CLAUDE.md`, **not** `AGENTS.md`, which is what some other harnesses look for. *(If you ever end up with both, put `@AGENTS.md` on the first line of your `CLAUDE.md` and it pulls the other one in.)*
- A room needs a known file for the map to **point at** — that's what `CONTEXT.md` is. It won't auto-load; it gets read because the map *routed* the AI there. Having that one consistent entry file in every room is what makes routing work at all.

- And the whole room is fair game: the AI can use *anything* in the folder — drafts, examples, notes. `CONTEXT.md` is just the door you walk in through.

Now the payoff. Save a job posting as `job-posting.md` inside `02-a-job-appears/`, then say *"tailor my resume for this job."* Claude reads that room, pulls only the real wins from `01-my-work/`, and writes the finished resume into `03-applications/` — working from your real experience instead of making things up, because the truth has exactly one home. **One folder quietly became a little factory.**

What You Just Did

Say it out loud, because this is the part people miss while they're busy following steps.

You didn't learn a prompt trick. **You built structured input.** Identity, rules, source material, and a map of where things live — all standing by in plain text, loaded before you type a word. The AI shows up to every session already knowing the story.

That's the upgrade. Your outputs got better because your *inputs* got better, and inputs are the half of this everybody ignores. From here on, when something the AI gives you is wrong, your first question stops being *"how do I re-word this?"* and becomes ***"what couldn't it see?"***

That question is the rest of the guide.

Stand on a Giant's Shoulders: ICM

This structure isn't something I made up. Numbered rooms as stages, a routing map up top, a `CONTEXT.md` in each room, the folder *itself* holding the state — don't worry about memorizing that list, you just built most of it. It has a name: the **Interpretable Context Methodology (ICM)**, from **Jake Van Clief and David McDermott**. It's the backbone of how I run everything, and it's the single best thing you could go learn next. Jake teaches it out in the open at **Clief Notes** — go learn it from the source (links in *Further Reading*).

What I add on top is two things: a way of *seeing* why it works — that you're writing a kind of language, a **soft DSL** (Part 4) — and this plain-English on-ramp so a non-coder can start today. Credit where it's due; the foundation is theirs.

PART 3 RECAP

- **Workspace root** = the folder you launch the agent in. **Rooms** = the folders inside it, one job each.
- You built a real workspace (`my-career`) — `CLAUDE.md` plus a few honest files — and pointed your agent at it.
- You made the leap: one folder → **rooms**, with a routing map up top.
- **Context is king**, and `CLAUDE.md` is its root: only `CLAUDE.md` auto-loads — every other file is read *only* because something reachable points to it.
- What you actually built is **structured input**. Better outputs, same model.
- That structure is **ICM** (Van Clief & McDermott) — go learn it from Jake's *Clief Notes*. Part 4 shows *why it's a language*; Part 5, *why the structure works*.

PART 4 OF 9

Markdown Is the DSL

The idea that ties the whole guide together: what you're really writing, and the one place it stops behaving like code.

You're Already Programming

Programmers have a term: **DSL**, a *domain-specific language* — a small, specialized language built for one job. SQL is a DSL for databases. When you wrote those three files in Part 3, you used one too. The plain English in your `CLAUDE.md` *is* a kind of program. Markdown is its syntax. The folder is its structure. And the AI is the thing that runs it.

What you're writing is a *soft* version of a DSL — so I've started calling it a **soft DSL**. *Soft* because, unlike a real programming language, the grammar is loose (it's conventions — headings, lists, a few rules — not a strict parser), and the thing that "runs" it isn't a compiler following exact steps. It's a probabilistic model *interpreting* your words. Same job as a DSL; two ingredients swapped. *(I'm writing this idea up properly — more on that soon.)*

That's not just a metaphor, and it's not only my framing: there's a documented method built on this same insight, and we put it to work in the very next part. For now, here's the mapping, once, so it sticks:

In normal code...	In your AI workspace...
The language's syntax	**Markdown** (headings, lists, links)
Where code can "see"	**Rooms** — scope: a room only sees what's in it
The entry point	`CLAUDE.md` — read first, every time
Modules / functions	`CONTEXT.md` files — one per room
Control flow	**The routing table** — "for this task, read that file"
Libraries you import	**Skills** — pre-built recipes (Part 6)
Input / output	**MCP** — connections to outside tools (Part 6)
Background subroutines	**Subagents** — helpers that run their own side-jobs

Once you see it this way, the rest of the guide stops being a list of tips and becomes one coherent thing: **you are writing software in English, and the workspace is the codebase.**

The One Honest Caveat

There's a catch, and saying it out loud is what keeps you out of trouble.

A normal program runs the same way every time. This one doesn't. A function you call behaves identically on every run. A plain-English instruction is *interpreted* — and the interpreter here is a language model weighing probabilities, not a compiler following exact rules. The same prompt can give different results. Karpathy memorably calls these models "people spirits" — brilliant, but fallible. This is the *soft* in soft DSL — and it's the one place the comparison to real code genuinely breaks.

So English **steers** the AI; it doesn't **compile** to guaranteed behavior. A rule in your `CLAUDE.md` makes good behavior *much more likely* — not certain. That single fact is *why* the rest of this guide matters: because the interpreter is fuzzy, you reduce what it has to guess. You give it the right context and nothing extra. You keep a human in the loop to check the output. You're not writing rigid code — you're writing clear instructions for a brilliant, slightly forgetful collaborator, and setting things up so it's easy for them to get it right.

PART 4 RECAP

- Plain English in markdown is a real language for directing AI — I call it a **soft DSL**: folders are the structure, the AI is the interpreter.
- It isn't just a trick — there's a documented method built on this exact idea (next part).
- The interpreter is *probabilistic*. English steers; it doesn't compile. Design for that.

PART 5 OF 9

Workspaces & Routing

Buckle in. Why the structure works — and how to design a workspace of your own instead of copying mine.

Read This Part Twice

This one is conceptual — no build steps, just the *why*. And it's the part that pays for the whole guide, because once you understand *why* the structure works, you stop copying my folders and start designing your own.

That's the goal here. Not "use Tate's template." **You should walk out of this part able to look at any messy thing you do over and over and lay out a workspace for it.** It's free-form: no schema, no validator, no correct answer. Just rooms that make sense to you, a map that points at them, and a reason behind every choice. It takes some real thinking the first time.

Tokens, and Why This Matters

First, the unit everything is measured in: the **token**. A token is a chunk of text — very roughly **¾ of a word** (so 100 tokens ≈ 75 words). Common words are one token; long or rare ones split into pieces (*"hamburger"* is a few). Exact counts vary by AI, but the rule of thumb holds.

Everything the AI reads, everything you type, everything it writes — all tokens. It can only hold so many at once, in what's called its **context window**. Modern windows are huge — today's leading models hold around a **million tokens** — so you'd think you could just dump everything in.

You can't, and here's the part most people miss: **a big window doesn't mean even attention.** There's a well-documented effect — researchers call it *context rot*, or being *"lost in the middle"* — where the more you stuff in, the worse the AI gets at using any of it, even on easy tasks. It reads the start and the end well and lets the middle go fuzzy.

> **Picture it.** Someone hands you a 400-page binder and says the answer's in there. Now picture being handed the four pages that matter. The information available is identical — but your odds of finding the answer, and your odds of getting derailed by something on page 231, are completely different. That's the AI, every session. **Structure is how you hand it the four pages.**

Which means **what the AI reads matters as much as what you ask.** That's why workspace structure is the single biggest upgrade you can make. Not because you'll run out of room — because *less, but relevant* beats *everything, just in case*.

The Method in Full: ICM

You met this method in Part 3: **ICM** — the *Interpretable Context Methodology* of Jake Van Clief and David McDermott (2026). There you *built* it — rooms and a routing map. Here's *why the structure works*, and how to design your own from scratch. Their paper puts the whole idea in three sentences:

> **"Stage sequencing is the folder numbering. Context scoping is the folder hierarchy. State management is the files on disk."**

Read that again. Every job a piece of software would normally do — order the steps, decide what's in scope, remember where things stand — is being done by **the folder itself**. Not by code, not by a chat you'll lose. The soft DSL from Part 4 is the language; ICM is the method written in it. Everything below is that method in depth.

The 3-Layer System

Instead of one giant conversation or one bloated `CONTEXT.md`, you split work into rooms, each for a different kind of work. Three layers:

- **Layer 1 — The Map (`CLAUDE.md`).** Top of the workspace, read first, every time. It says what the project is, what the rooms are, your naming conventions, and — most importantly — it

carries a **routing table**: for a given task, here's the file to read, and (just as important) here's what to *skip*.

- **Layer 2 — The Rooms (`CONTEXT.md`).** Each room has its own. When the AI works in a room, it reads *that* room's context: what this room is for, the process, what lives here. A few plain-English paragraphs.
- **Layer 3 — The Tools (Skills).** Pre-built recipes you plug into a room. Each room loads only the ones it needs (Part 6).

Underneath the layers sit **ICM's five design principles.** These are the paper's, in its own words, with my gloss — steal them outright:

- **One stage, one job.** A room does one thing. If yours is doing two, it's two rooms.
- **Plain text as the interface.** Rooms hand work to each other as plain markdown files. (The paper traces this straight back to Unix — text as the universal interface between programs.)
- **Layered context loading.** Shallow at the top, deep only where needed. Route, don't dump. *This is the one that beats a bigger context window.*
- **Every output is an edit surface.** Whatever a step produces is a file you can open and fix before the next step runs — a review gate at every boundary.
- **Configure the factory, not the product.** You're not making one thing; you're setting up the line that makes all of them.

Two more I'd add from running this daily, and they're mine, not theirs: **say what to skip**, not only what to read — naming the exclusions is what actually keeps context narrow. And **write about the work, not about the AI.**

Why Routing Saves You (a worked example)

This isn't theory for me — it's how I actually run TerraByte. Two real examples:

The Socials workspace is a stage-first assembly line, not a folder-per-platform. A post moves through rooms — `01-writing-room` → `02-content-creation` → `03-distribution` — and only takes on a platform shape (Instagram, YouTube, TikTok) at the very end. When I say *"let's draft this week's reel,"* the AI reads the map, routes to the writing room, and ignores everything about distribution. It isn't holding five platforms' rules in its head while we write one script.

The Resources workspace (where this very guide is being built) runs `01-brainstorming+planning` → `02-building` → `05-library`, with a reference shelf and a toolbox off to the side. Each resource carries its own status in its own file, so any session — me today, me in a month, or an AI picking it up cold — knows exactly where it stands.

What a Workspace Actually Looks Like

Here's that Resources workspace drawn out — the real thing, simplified just enough to read:

```
my-resources/                    ← the workspace root
├── CLAUDE.md                    ← read first, every time
│
├── 01-brainstorming+planning/ ← stage 1: shape the idea
│   └── CONTEXT.md
├── 02-building/                 ← stage 2: actually make it
│   └── CONTEXT.md
├── 05-library/                  ← stage 5: the finished shelf
│   └── CONTEXT.md
│
├── 03-documentation/            ← reference (brand & voice)
│   └── CONTEXT.md
└── 04-utilities/                ← templates & scripts
    └── CONTEXT.md
```

The numbers aren't decoration — they're the *order of the work*. An idea starts in `01`, gets built in `02`, and ends up on the shelf in `05`. The off-by-themselves rooms (`03`, `04`) sit out of the numbered flow on purpose: every stage reaches into them, so they don't belong to any one step. You can read the whole process just by reading the folder names. **That's the idea in one picture — the folder structure *is* the architecture.**

The room contract

Every room holds one short file — its `CONTEXT.md` — that works like a contract for that room. (This five-field shape is my own room template — a fuller take on ICM's shorter stage contract.) A good one answers five things:

- **Identity** — what this room is for, in a sentence.
- **Inputs** — what arrives here, and from where.

- **Process** — the steps that happen inside.
- **Outputs** — what leaves, and in what shape.
- **Handoff** — which room it goes to next.

Keep it short. If a room's contract grows past a page, that's usually the room doing two jobs — split it. (Short isn't just tidiness: the AI reads this file *every* time it works in the room, so every extra line is budget spent on overhead instead of work.)

Naming, So Files Don't Pile Up

You already built a routing map in Part 3 — name the rooms, point the AI at the right one, and (just as important) tell it what to *skip*. The last piece of a good map is a **naming convention**, so files stay findable by pattern instead of memory:

- Drafts: `topic-name_draft.md`
- Finals: `topic-name_final.md`
- Published / dated: `YYYY-MM-topic.md`

With a map and a naming rule, *"work on this week's reel in the writing room"* is all it takes: the AI reads the map, opens one room, skips the rest, and already knows where the new file should land. Focused, cheap, and it never loses your work.

State Lives in the Folder

Here's the quiet superpower — the third of those three sentences, *state management is the files on disk*: **where a project stands lives in a file, not in your memory or a chat log.**

The simplest version — a status line at the top of whatever you're working on:

```
# This Week's Reel — script

STATUS: drafting   (drafting → review → final)
Last updated: 2026-06-18
```

That one line means any session can pick the work up cold — including you, three weeks from now, with no memory of where you left off. It's the same reason I keep a short, dated log in my project files: the project carries its own story so I don't have to.

This is also the answer to *"do I rebuild my setup for every project?"* No — this is the *configure the factory, not the product* principle in practice. **You set the workspace up once and reuse it:** the rooms, the routing, the naming. Then you just work inside it. Build the factory once; it keeps producing.

And keep it alive. Projects evolve, and context that doesn't evolve with them quietly goes wrong — **wrong context is worse than no context**, because the AI trusts it completely. Five minutes a week keeping your `CONTEXT.md` files current is the single highest-leverage habit in this whole guide.

Designing Your Own — Four Steps

1. **List your rooms.** The 2–3 major modes of your work. *Rule of thumb: if you've ever wished the AI would "forget what it was just doing," that's a room boundary.*
2. **Write a `CONTEXT.md` for each** — its Identity, Inputs, Process, Outputs, and Handoff. Keep it under a page.
3. **Write your `CLAUDE.md`** — list the rooms, build the routing table, add naming conventions so files stay organized by pattern, no code required.
4. **Start working,** then adjust your context files based on what the AI gets right and wrong. The system gets better every iteration.

Nothing about those four steps is specific to me, or to writing, or to job hunting. Point them at a side business, a research habit, a class, a house renovation — the method doesn't care what the work is. **That's what you take with you.**

PART 5 RECAP

- This method has a name — **ICM** — and five principles, led by *one stage, one job* and *layered context loading*.
- Tokens are the budget; a bigger window doesn't fix *context rot*. Relevant beats everything.
- Map (`CLAUDE.md`) → Rooms (`CONTEXT.md`) → Tools (Skills). Numbered rooms are the *order of the work*.
- Each room is a contract: Identity · Inputs · Process · Outputs · Handoff. Keep it short.
- **State lives in files**, not your head — a `STATUS:` line lets any session resume cold. Keep it current; stale context is worse than none.
- Route to what's needed; say what to skip. Start with 2–3 rooms — and design your own, for your own work.

PART 6 OF 9

Skills, MCP & Subagents

The layer that turns "good prompts" into "a system that does the work." Three ideas, universal across every harness.

These three are worth knowing by name, because they're not Claude Code trivia — **the same three ideas show up in every agent harness**, and they mean the same thing in each one. They also get mixed up constantly, so here they are plainly.

Skills — recipes the AI can follow

A skill is a pre-packaged recipe. Someone figured out the best way to, say, audit a landing page, wrote the steps in a file, and now the AI can follow them on demand. Technically a skill is just a `SKILL.md` file — a bit of markdown with instructions — living in a `skills` folder (`~/.claude/skills/` for personal ones, `.claude/skills/` inside a project). Some ship built in already. You can write your own, no coding required — it's the same plain-English markdown you've used all along. And because there's a shared open spec behind them, a well-written skill travels: the same skill folder works in Claude Code, Codex, and the rest — only where you drop it changes.

(You'll see skills shipped inside "plugins," "packs," or "extensions" depending on the harness. That's packaging, not a new concept — one bundle that can install several things at once.)

MCP — plugging in your other tools

MCP is the plumbing that lets the AI talk to your outside tools — Google Drive, your calendar, a database, GitHub. Instead of copy-pasting data in, you connect the tool once and the AI can read and act on it directly. (Under the hood each connection also defines exactly which actions the AI is allowed to take — it's a doorway with a specific set of doors, not a blank check.) You don't need this on day one; just know it's how the system reaches the rest of your stack when you're ready.

Subagents — helpers for side-jobs

A **subagent** is a helper the AI hands a side-job to — research, a review, a chunk of grunt work — that runs in *its own* fresh context and reports back. Why it matters connects straight to Part 5: the side-job's clutter never fills up your main conversation. It's the routing principle again — narrow context — but for a whole task instead of a file.

The Beginner Rule

Add one thing at a time. Install one skill, test it on a real task, see if the output feels right — *then* add another. Stacking five on day one is how you lose the plot: when something goes wrong, you won't know which one caused it.

PART 6 RECAP

- **Skill** = a recipe (`SKILL.md`) the AI follows. Bundles of them get called plugins or packs — same thing.
- **MCP** = connections to your outside tools and data.
- **Subagent** = a helper that does a side-job in its own context.
- All three are universal across harnesses. Add one at a time; test before stacking.

PART 7 OF 9

The 8 Common Mistakes

Most problems trace to one of these. Skim now; come back when something feels off.

1) Making `CLAUDE.md` too long. Burns tokens, muddies context, and buries the routing table. *Target: one page, max.* If it's growing, that content belongs in a room's `CONTEXT.md`.

2) Skipping the routing table. Without it, the AI guesses what to read or opens everything — random or wasteful. Keep it simple: Task, Go-to, Read.

3) Never saying what *not* to load. The flip side of #2, and the one almost everyone misses. "Read this" is half the instruction; "skip that" is the other half. Naming what to ignore is what keeps context narrow — and narrow context is the whole game (Part 5).

4) Too many rooms, too soon. Every room is a file to maintain and a mental shift for you. *Start with 2–3.* Add one when a real, repeated need shows up — not a theoretical one.

5) Writing context about the AI instead of about the work. *"Be creative and bold"* is nearly useless. *"Our audience is CTOs at 50–500-person companies; they've heard every pitch and respond to concrete numbers, not adjectives"* changes the output. **Aim for 80% about the work, 20% or less about behavior.**

6) Never updating context. Stale context is worse than none — see Part 5.

7) Everything in one folder, hoping. When things feel tangled and the AI keeps asking the same questions, it's almost always a *structure* problem, not a *prompt* problem.

8) Building the whole system before using it. Don't design a perfect six-room architecture before you've tried a simple one. **Start rough. Let real tasks tell you what to add.** Perfection is a trap; iteration is the move.

PART 8 OF 9

Cheat Codes

Five things nobody tells you, that every serious user turns on in week one.

1) Use plan mode. Always.

Every harness has one — Claude Code, Codex, Antigravity, Grok Build. Turn it on and the agent **stops trying to do the thing** and instead asks you questions first, then writes out what it's about to do so you can approve or redirect it.

Here's why it's the best habit in this whole guide: that back-and-forth *is* prompt engineering, done for you. You describe something half-formed, it asks the three questions you hadn't thought about, and the answer to those questions becomes the real instruction. You get a better prompt without writing one — and you catch the misunderstanding before it writes twelve files instead of after.

Plan first, execute second. For anything bigger than a one-liner, it costs you sixty seconds and saves you a rewrite.

2) Turn off the permission prompts (once you trust the room)

Every harness ships a *"yes, I know what I'm doing"* mode — auto-accept, always-allow, YOLO mode, or in Claude Code, `--dangerously-skip-permissions`. It stops the agent asking you to confirm every single file edit.

I'll be straight about it: **it looks like a security gap because it is one.** You're handing the agent a blank check inside everything it can reach. And essentially everybody doing real work turns it on anyway, because approving an edit every two minutes destroys the thing that made this worth doing.

So use it like a grown-up: turn it on inside a workspace whose contents you could lose without crying, keep it off when the agent can reach your whole drive, and have a copy of anything you'd hate to redo. Your undo button is your backup, not the agent's good intentions.

3) Your subscription is more than one model

It's not one AI — it's a lineup. There's usually a fast one and a deep one, and switching is a single command (`/model` in Claude Code). Most people never touch it and quietly use the wrong tool all day.

The rule of thumb: **fast model for mechanical work** — reformatting, renaming, filling in a template, tidying a file. **Deep model for anything you'd have to double-check** — architecture, tricky reasoning, the first draft of something that matters. Try the same task on both once. You'll feel the difference immediately, and you'll stop burning your good model on busywork.

4) Ask your agent what you're already paying for

The most underrated prompt there is:

> *"What can you actually do that I probably don't know about? What's included in my plan?"*

Your subscription almost certainly bundles tools, limits, and abilities you've never opened — and your agent knows its own capabilities better than any article does. Ask it, then go poke at whatever comes back. Half of getting good at this is just discovering what was already sitting there.

5) Map files cascade — so keep the top one light

This one bites people months in, so learn it now.

Your harness doesn't read *one* map file. It walks **up** the folder tree from wherever you launched and loads every map file it passes — the one in your home folder, then the one above your project, then your project's own, closest last. They stack.

Which means a `CLAUDE.md` sitting in your home folder is loaded into **every single session you ever start**, forever, in every project.

So split them deliberately:

- **Top level (home folder):** only what's true everywhere. How you like to be talked to. Your hard rules. A dozen lines, not a hundred.
- **Workspace level:** everything about *this* project — the rooms, the routing table, the naming, the one rule.

If a fact only matters in one workspace, it does not belong at the top. A bloated top-level map is a tax you pay on every session for the rest of your life, and it's the fastest way to quietly poison good context with irrelevant context.

PART 9 OF 9

Launch

Two days to confident. Then a challenge.

Two Days, Not a Month

You don't need a week of study for this. Everything in Parts 1–3 is about two and a half hours of real work, and you should do it now, while it's fresh. Here's the honest schedule:

Day 1 — Stand it up. *(~90 minutes)*

1. Subscribe, install your harness, install Obsidian. *(20 min)*
2. Make one folder for a **real** project — something you're actually working on this week, not a sample. You'll only feel the system click on something you care about. *(1 min)*
3. Write `CLAUDE.md` and one `CONTEXT.md` from the templates in Part 3. Under a page each. Clumsy is fine — it's supposed to be. *(30 min)*
4. Launch your agent in that folder and give it one real task you had to do anyway — in plan mode (Part 8). *(30 min)*
5. Fix one thing in `CLAUDE.md` based on what it got wrong. *(10 min)*

Day 2 — Make it a workspace. *(~60 minutes)*

1. Split the folder into 2–3 rooms — the stages your work actually moves through. *(20 min)*
2. Give each room a `CONTEXT.md` contract: Identity · Inputs · Process · Outputs · Handoff. *(20 min)*
3. Add the routing table to `CLAUDE.md`, and say what to **skip**. *(10 min)*
4. Run another real task and watch it route itself. *(10 min)*

That's it. You're not "studying the system" for a month — you're using it on day one and tuning it forever after. Two days from now you should be building *with* AI, not reading about it.

Where to Go From Here

You've got the wiring. The next mile is the ecosystem it plugs into — take these in whatever order your work actually demands:

- **Go deeper on ICM.** Jake Van Clief teaches it at *Clief Notes* — the method itself, from the source. This is the highest-value next step by a wide margin.
- **Live in Obsidian.** Links between notes, tags, daily notes, plugins. The better you get at organizing plain text, the better every workspace you build gets.
- **Learn GitHub.** Version control is how you keep a history of a workspace, roll back a bad change, share one, or work on it with someone else. It's also the single biggest unlock if you ever want the AI to build real software with you.
- **Add one tool at a time.** A skill, then an MCP connection, then a subagent (Part 6). One, tested, then the next.

If You Get Stuck

- **The TerraByte Discord** — `discord.gg/p9TVXqfjxD` — the fastest way to reach me, one on one. Bring the workspace you're stuck on; someone in there has hit the same wall.
- **@terrabyte.dev on Instagram** — quick tips and short breakdowns.
- **terrabytesolutions.com** — everything else I build.
- **Clief Notes** — `skool.com/cliefnotes` — where ICM is taught and argued about.

One Last Thing

It gets easier every week. Your first `CLAUDE.md` will feel clumsy. Your second will be better. By the fourth, you'll wonder how you worked without one.

You don't need to be technical. You need to be clear. That's a skill you already have — you're just pointing it at a new kind of language.

THE CHALLENGE

Build one workspace this week. Run one real task through it. Then show me.

Not a demo folder — the messiest recurring thing you do. The one you'd describe to a friend as *"ugh, every time I have to..."* Give it a workspace root, two or three rooms, a map with a routing table, and one honest `CONTEXT.md` per room. Point your agent at it. Hand it the task you were going to do anyway.

Then screenshot the folder tree and drop it in the Discord — `discord.gg/p9TVXqfjxD`. I want to see what you named your rooms, and so does everyone else in there.

You'll know within one task whether it worked. That's the whole game.

Further Reading

- Van Clief, J. & McDermott, D. (2026). *Interpretable Context Methodology: Folder Structure as Agentic Architecture.* arXiv preprint 2603.16021. — the method (ICM) this guide is built on.
- **Clief Notes** — Jake Van Clief's community, where ICM is taught and discussed: `skool.com/cliefnotes`. The best place to go deeper on the method.
- **Soft DSL** (TerraByte, forthcoming). The idea behind Part 4 — that structured markdown is a new kind of *soft* domain-specific language: loose grammar, learned interpreter. I'm writing it up; ask me for it when it's out.
- Karpathy, A. (2023). *"The hottest new programming language is English."* — the origin of the English-as-programming idea.
- Karpathy, A. (2025). *Software Is Changing (Again)* (Y Combinator AI Startup School, June 2025). — the "Software 3.0" talk: LLMs as a new computer you program in plain language.
- Hong, K., Troynikov, A. & Huber, J. (2025). *Context Rot: How Increasing Input Tokens Impacts LLM Performance.* Chroma. — plus Liu et al. (2024), *Lost in the Middle,* TACL 12:157–173. Why a bigger context window isn't even attention.