Context by Design

A Design Manual for Your AI Work OS

Omar Shraim

TECHNOLOGY EXECUTIVE · AI STRATEGY & DIGITAL TRANSFORMATION

omarshraim.com

Context by Design: A Design Manual for Your AI Work OS

Edition: v14, July 2026. What changed in this edition: the edition line you are reading was added, the Chapter 11 build-partner prompt now reports the guide's edition and checks for a newer one, and Chapter 7 states plainly that the walled vault's guidance is reasoned while the one-principal vault's is lived.

This guide describes a way of working with AI agents that is tool-independent in principle. Product and tool names are used only as current examples.

Cedarline Advisory, Medence, and every person, firm, and event portrayed in this guide's scenes are fictional or fictionalized composites. Any resemblance to real persons, organizations, or engagements is coincidental.

AI output is an input to professional judgment, not a substitute for it. Nothing in this guide replaces legal, financial, or other professional review.

How to Read This Guide

Context by Design: A Design Manual for Your AI Work OS

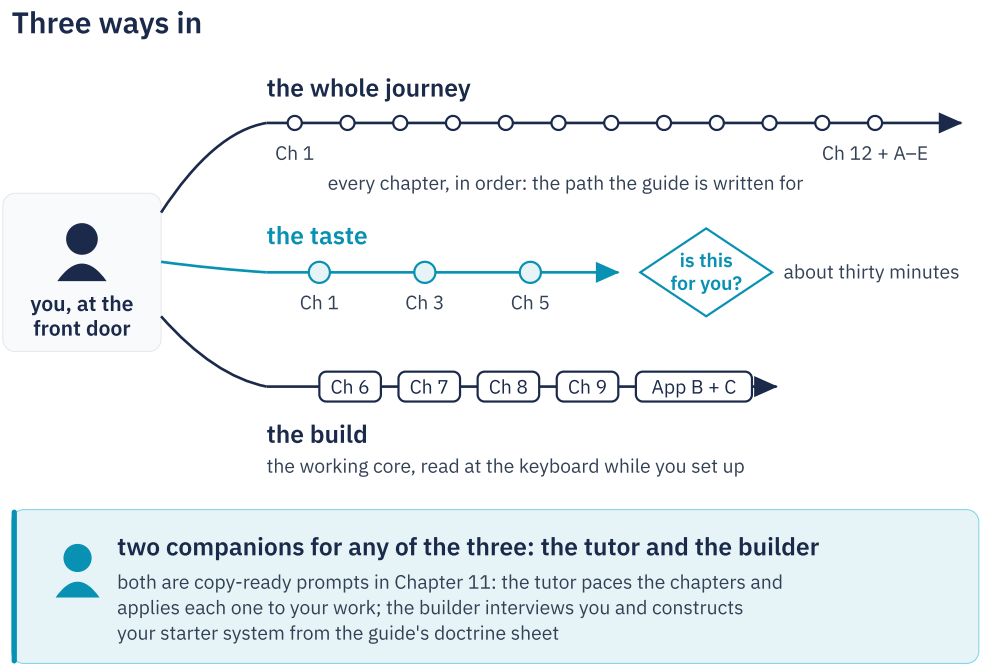
This is a design manual for the individual's AI work environment: how one professional builds the folders, files, and standing rules that let an AI agent work from the full picture of their work, instead of starting from zero every session. It is three books in one. An **argument**: why context, not the model, is where the value sits (Chapters 1 to 4). A **build**: a working implementation with today's tools, from setup to the applied systems built on top (Chapters 5, 6, 9, 10, 11, and the appendices); the matured skills it points to are ones you grow yourself, from prompts the guide prints in full. A **doctrine**: the design rules and trade-offs that outlast any tool (Chapters 7, 8, and 12).

It is written for the knowledge professional, not the developer: if your work lives in documents, email, meetings, and decisions, this guide assumes no coding background and was written for you. What it does assume is a willingness to follow setup steps, in a desktop app at first and a terminal as the system matures, with every step printed or pointed to. It is not a book of prompt techniques, not an enterprise rollout playbook, and not a manual for building agents as software; a reader hunting any of those will be better served elsewhere.

It is a guide first and a reference second, written to be read in order, and it opens with a week inside a fictional firm, Cedarline Advisory, so you can see what the system buys before you build it. The next page maps the ways in.

Not every reader should take the same door, so here are three. **The whole journey**: start to finish, in order, the way the chapters were built to be read. **The taste**, for the reader still deciding: Chapter 1 for what the system buys, Chapter 3 for what an agent actually is, Chapter 5 for how little it takes to start; about thirty minutes, then decide. **The build**, for the reader already convinced and sitting at the keyboard: Chapter 5's setup notes first, then Chapters 6 through 9 plus Appendices B and C, the working core, read while setting up.

Two companions fit any of the three doors. Read with a **tutor**: hand this PDF to an AI collaborator with the copy-ready tutor prompt in Chapter 11, and it will pace the chapters and connect each one to your own work. Or put the book to work with the **builder**: Appendix E is the doctrine sheet, a distillation of the guide's rules written to be read by an AI collaborator and regenerated with every edition, and the builder prompt in Chapter 11 has that collaborator interview you and construct your starter system from it. This book can help construct what it teaches.



Three ways into the guide, and two AI companions, the tutor and the builder, that fit any of them.

One image to carry throughout: the system is a ladder of layers, each added only when the work asks for it, standing rules, then running state, then cross-folder context, then live channels, then memory. You do not build the ladder in a day; Chapter 12 ties the climb off. Once read, the guide doubles as a reference: the table of contents on the next page lists every chapter with its page number, and the five appendices, a worked example, two setup procedures, a concordance of the field's tangled vocabulary, and the doctrine sheet, sit off the main reading path for when you need them.

Contents

Chapter 1: Your First Week at Cedarline Advisory

You start on a Sunday, because the working week in the Gulf starts on a Sunday. Cedarline Advisory is a mid-sized advisory firm: a few hundred people, offices in the Gulf and the Levant, a practice that runs on contracts, negotiations, board papers, and the judgment of a handful of senior people who carry far more in their heads than any one person should. You are the newest hire. Nobody has told you yet what your job really is. For the first week, your job is to watch.

You shadow four people.

Ken is the Deputy Managing Director, and his remit has long since spilled past his title. He sits where commercial, legal, and delivery overlap, which means he catches the things that fall between them. **Lama** runs Commercial. She lives in negotiations, often three at once, and she is the calmest person in any room where money is being argued over. **Amer** is a strategy lead who is quietly building a name in AI, reading constantly, watching the right people, trying to turn a hundred scattered inputs into something that is his own. **Roni** is the chief of staff who runs the firm's daily cadence, and is somehow always already briefed when everyone else is still opening their laptops.

They do not work the way you expect. There is no wall of dashboards, no team of analysts. Each of them works out of plain folders on a laptop, talking to an AI agent that already seems to know the work. You will spend the rest of this guide learning to build what they have. This week, you only need to see what it buys them.

Here are three moments.

The clause nobody else caught

The contract was ninety-one pages and Ken had read every one of them twice. So had Legal. So had Finance. The number everyone had agreed on was a hundred thousand. A manageable bet. He had said as much in three meetings.

Legal had read it for the legal terms. They had assumed, the way everyone always assumes, that the commercial shape of the deal belonged to the delivery team. The delivery team had

read the parts that touched delivery. Nobody owned the whole. The gap was not a mistake anyone made. It was the space between three people who each read their own part well.

It was nine at night when Ken asked the agent to read the signed copy from the top. Not the summary. The actual document, sitting in the folder, the whole of it, at once. He told it to read for the commercial exposure, not the clauses, the money: term, rates, the way the years actually added up. Then he went to make coffee.

When he came back there was a line waiting for him. The term was five years, not two. The Year 3 rate stepped up and stayed there. Add the years the way the contract actually counted them and Cedarline was not exposed to a hundred thousand. It was exposed to one-point-one million, and it was enforceable. Two clauses underneath made it worse: one had the firm on record as knowingly buying the thing it was now disputing, the other put adoption squarely on Cedarline's side of the line.

Nobody had hidden it. It was spread across ninety-one pages, in three sections, in the kind of language built to be read and not understood. The agent had the whole document in front of it at once, and it added up.

He did not send the email he had drafted. He asked the agent one more thing: explain the lawyer's letter to me in plain words, the parts I half-know, and help me answer it in language I can stand behind in a meeting. Then he wrote a different email.

You could have uploaded the contract to a browser chatbot and asked it good questions. What no chat window had was the rest: six months of the deal's history, the positions already taken, the sensitivities already flagged, sitting beside the document in one folder the agent reads as a matter of course. Ken's agent had the whole picture because the whole picture lived in the folder. You will build that folder in Chapters 7 and 8, and the file that carries the deal's memory in Chapter 6.

The comment that went to the wrong room

Two matters, two folders, two separate conversations open on the same screen. One was a lawsuit that could run for a year. The other was a contract Ken was trying to close by Friday. He had been moving between them all afternoon, and somewhere in the fourth hour he typed a sentence meant for the contract into the litigation window and hit send.

The agent did not run with it. It stopped. *This seems off track for the litigation. It reads like it belongs to the contract negotiation. Did you mean the other folder?*

He sat back. It was right. He had crossed his own wires, and the system had caught him, because it knew which room he was standing in and what that room was about. He moved

the comment where it belonged.

It was a small thing. It was also the whole point. Every folder he had built, every standing-rules file, every bit of discipline he had grumbled about setting up, had just paid for itself in one sentence.

This is the part a single chat thread cannot do, because a single chat thread has no rooms. The agent caught the stray comment because each folder is a separate workspace with its own context, its own rules, its own job. Building those rooms is what most of this guide is about. The mechanics are in Chapter 7.

The second brain

Amer had been trying to build a name in AI for two years, and for two years it had evaporated. He read constantly. He watched the right people. A highlight here, a note there, a screenshot he never found again. None of it added up to anything he could call his own.

Now every worthwhile read and every worthwhile video goes into one folder. The agent pulls the transcript, checks the new material against what he already knows, and grows a private wiki that cross-references itself. Six months in, when Amer sits down to write, he is not starting from a blank page. He is drawing on a body of his own thinking, dense enough that it has started to connect ideas he never connected himself.

You should know one thing about Amer's folder, because it is the thing the internet gets wrong. What Amer built is the "second brain" everyone online is selling. It is real and it is useful. But it is one room in the firm, not the firm. It accumulates knowledge; it does not run the work. Ken's contract folder and Lama's negotiations are a different kind of system, and the confusion between the two is the fog this guide exists to clear.

Six months of Amer's reading compounds because it lives in files he owns, cross-referenced where he can see them, not scattered through old conversations or a product's memory of them. The compounding is the entire value. You will build the knowledge base in Chapter 11, and learn to see it as one room among many in Chapter 4.

None of these four people is more gifted than you. They are not faster typists or harder workers. They built a system, one folder at a time, that holds the context of their work so the agent can act on the whole picture instead of starting from zero every morning. That system is the subject of this guide, and by the end you will have built your own.

One honest note before you start, because you are already thinking it: *couldn't I do all of this with ChatGPT, or any chat window?* You could try, and the chat products are trying to meet you: they now remember some of what you tell them and hold documents in project spaces.

But in each of these scenes, the thing that made the difference was not a cleverer answer. It was that the agent was working inside a folder that already held the document, the history, the rules, and the relationships, a place you built, can read, and can correct. What a chat product remembers is chosen by the product and held by the vendor; what a folder holds is chosen by you and lives on your own disk, where any tool can read it. The rest of this guide is about building the place the context lives.

One boundary also stands from the first scene onward, and it holds through every example in this book. The agent's reading of the contract informed Ken's judgment and sharpened his questions; it did not replace his lawyer. Nothing an AI produces in this system substitutes for legal, financial, or other professional review, and every consequential action, a signature, a payment, a position taken in a dispute, stays behind a human check. The system this guide builds briefs the professional. It does not stand in for one.

Chapter 2: The Reader and the Problem

Who This Is For

Summary: This guide is for the knowledge professional, in any size of organization, who wants agentic AI to grow their capacity, efficiency, and effectiveness, not to write software.

Setting the scene

Before the problem, the reader. This guide is written for the person whose work is knowledge work inside a business or operations context: decisions, documents, communications, relationships, and judgment. It does not matter whether you sit in a large corporation, run a small startup, or work for yourself. What matters is that your work is recurring, it carries history, and you want AI to help you carry more of it: more capacity, more efficiency, more effectiveness, without hiring a technical team or becoming technical yourself.

That framing matters because it decides whether the rest of this guide is for you. The problem the next chapter describes, and the system this guide builds to solve it, only becomes valuable when your work is continuous and context-heavy. If you use AI for occasional one-off questions, you will not feel the gap. If AI is becoming part of how you actually operate week to week, you will feel it constantly, and this guide is the response.

The persona: the high-context professional

Most online AI content targets developers and “second brain” hobbyists. The mainstream of business and operations work, the people who live in email and documents and run their work through judgment and relationships, has almost nothing written for it. A common first reaction to agentic AI tools from this audience is “I thought those were only for coding.”

It helps to name this reader: the **high-context professional**. Their work is high-context by nature. A negotiation carries six months of stakeholder history. A tender response reflects commercial sensitivities, vendor positioning, internal approval chains, and prior commitments. An executive email requires knowing the recipient, the relationship, and what was said in three prior meetings. None of that fits in a chat prompt, yet all of it must be available to an AI to produce anything worth using.

“High-context” also names the precise failure mode with standard AI: the tool returns generic, context-free output because it was never told the actual context. The system this guide builds is the fix.

Profile

- Works across multiple domains at once: strategy, commercial, operations, communications, product, stakeholder management.
- Primary outputs are decisions, communications, and influence, not code and not notes.
- Lives in email, documents, calendars, meetings and their transcripts, and team chat as the main surfaces of work.
- Technically literate but not technical: adopts tools when the value is clear, has no interest in managing infrastructure.
- Often multi-role or multi-organization, common from solo operators to the C-suite.
- Work pauses and resumes across days and weeks, so context continuity is a real operational cost.
- Friction is expensive: any system that demands ongoing maintenance gets abandoned.

Two practical prerequisites belong in this profile, stated once and honestly. First, the system requires installing software and connecting accounts, so it fits the reader who has that authority over their machine; on a managed corporate laptop, IT approval may have to come first. Second, the work this system will hold is often an employer’s or a client’s: confirm you are authorized to bring that material into an AI-reachable workspace, and use an account tier consistent with your organization’s rules. The vault boundaries in Chapter 7 and the routing discipline in Chapter 10 build on that authorization; they do not replace it.

The reason this reader is underserved is not that the use case is a minority. It is that the people writing the content are not them.

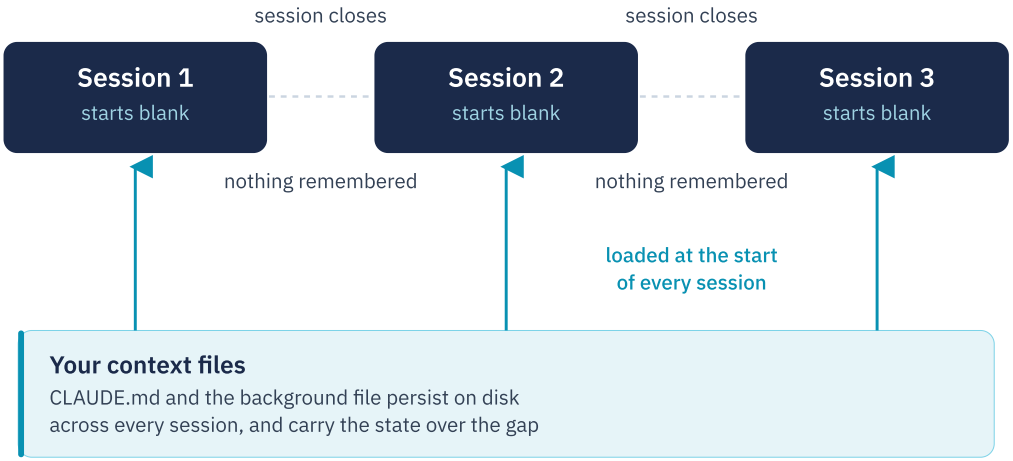
The Stateless Problem

Summary: Every AI session starts from zero by design, and that architectural fact becomes a serious friction point the moment AI is applied to recurring, evolving knowledge work.

Every large language model session is stateless. When a session closes, the model retains nothing: no memory of what was discussed, decided, or established. The next session starts with a clean slate. This is no vendor’s defect and no design flaw. The architecture was built for stateless, single-session interactions, and persistent memory across sessions is not built in.

The products wrapped around these models increasingly compensate: browser chatbots now offer memory features and project spaces that carry some context between conversations. Be clear about what that buys and what it does not. What those features retain is selected by the product and held on the vendor's side, where you cannot see all of it, scope it to one engagement, correct it with confidence, or take it with you to another tool. For the recurring, sensitive, multi-project work this guide is about, that is not a briefing system; it is a black box with a memory. The fix this guide builds is the same externalization, done where you can govern it: in files you own.

A useful image is the 2004 film *50 First Dates*. Lucy resets her memory every time she sleeps and wakes with no recollection of the previous day. Henry compensates with a video tape, a scrapbook, and a daily re-introduction. The folders and files this guide builds are that scrapbook. The agent wakes at the start of each session with no memory of the work, and the context files are what you hand it so it can resume where you left off.



Every session starts blank; your context files persist on disk and carry the state across the gap.

The limitation is invisible for one-off tasks: answer a question, draft a message, summarize a document. Single-shot interactions never expose it. It becomes costly the moment AI is applied to the recurring, context-heavy work described in Who This Is For. The implied context of that work is large: your role, your active projects, your current decisions, your stakeholder relationships, your tone preferences, your sensitivities. Re-briefing all of it at the start of every session is expensive, inconsistent, and easy to get wrong. Something settled last week gets re-litigated. A sensitivity the AI knew about is forgotten. Worse, you will occasionally forget to re-brief, and the AI will operate on stale or incomplete information without signaling that anything is wrong. The browser chat is a hotel lobby: functional, mostly anonymous, and whatever the concierge remembers about you is kept in the hotel's notebook, not yours. The file-resident system is your office: your files are on the

desk, your standing instructions are on the wall, and the assistant knows who you are before you speak.

Four things follow. The gap is architectural, and the memory features vendors layer on top do not close it for high-context work; every system that addresses it fully works by externalizing context into files the AI reads at session start. The AI does not gain memory, you gain a briefing system, and its discipline is to brief the model completely and consistently, every time, from durable files. The cost stays hidden until the work is recurring, where continuous, multi-week work pays the largest tax and earns the largest return from closing the gap. And stale context is more dangerous than missing context, because a model that briefs from an outdated file acts with false confidence, which is why keeping the files current is part of the system, not an afterthought.

Two Clarifications That Shape the System

Summary: Two boundary-setting clarifications before the disciplines: the system is folders and files, not any particular editor, and direct integrations are a later layer, not a starting requirement.

1. **The vault is the system; your Markdown editor is only a window.** A definition first, because the word recurs throughout this guide: your **vault** is the root folder of your work OS, the single directory that holds all your workspace folders, including the knowledge-base folder, and it is the middle of the three layers you will meet in Chapter 6, machine, vault, folder. The word is borrowed from the Obsidian community, and here it means nothing more than that root folder. The vault is plain folders and Markdown files on disk, and that is the whole system: the folders, the files, and the agent that reads them. To read and edit the files yourself you need a Markdown editor, and any will do, Typora, Obsidian, VS Code (the editor most AI tutorial videos are filmed in), or another; use the one you are comfortable in. Be equally clear about what you do not need: you do not need Obsidian to build a work OS, and starting your journey there can quietly mislead, because its links, tags, and graph views belong to the “second brain” tradition and add nothing to the AI workflow. The agent reads files, not graphs. Do not invest in building a second brain for its own sake; that is the wrong destination for this reader. The agent is the thinking partner, the folders are the memory, and the editor is just the window.
2. **Direct integrations remove the copy-paste moment.** When the agent can read your actual work surfaces (email, calendar, drive, team chat) directly, the friction of “let me paste this thread in first” disappears. That is removable by configuration, not by discipline, and it is best seen as a later evolution layer, not a starting requirement. Chapter 10, Connecting Live Context Sources, covers it in full.

Capturing and Supplying Context

Summary: One discipline with two sides, capturing the implied context of your work into durable files and supplying it to the AI at the right moments, turns what only you know into a portable, file-resident system that any AI can consume from the first message.

The deeper goal of this system is not to use one tool better. It is to build a **personal context management system**: an organized, file-resident structure that captures the implied context of your work so any AI you work with can act on the full picture instead of starting from zero.

The discipline behind it has two sides. The first is **capture**: externalizing the implied context of your work into durable, file-resident structures, then maintaining them as the work evolves. The key phrase is *implied context*: the things you know about your work that you never think to state because they are background assumptions. Your authority. The sensitivities around a deal. The standing decision that a vendor is not under consideration. The constraint a regulation introduced three months ago that now silently shapes every decision. Implied context is what separates a subject-matter expert from someone who just read the brief. An AI with no implied context is someone who just read the brief, every session.

The second side is **supply**. Where capture asks “what do I need to write down?”, supply asks “how do I load, provide, and maintain context within and across sessions?” It covers what to load at session start, how to bring context in mid-session, how to close a session so the next one starts well, and how to handle context across machines and over time. The two sides are inseparable. Capture without supply produces well-organized files that nobody loads; supply without capture produces good loading habits with nothing durable to load.

Picture the difference in one task. Ask a fresh chatbot to reply to a supplier’s email and it writes something competent and generic, because it does not know you decided weeks ago not to renew with them, that the relationship is strained, or that Legal asked you to keep everything in writing. Ask an agent briefed from your files the same thing and the reply carries all three, without you stating any of them. That gap is implied context made durable: the first system starts from the email, the second starts from everything around it.

None of this is an AI invention. These are professional knowledge management conventions that predate AI entirely. What changes is that the model becomes an active reader of your knowledge system, consuming the structure at the start of every session, which gives you a powerful incentive to apply conventions you should have been applying anyway. And the payoff outlasts any one tool: the file-resident system is a structured, plain-text representation of your professional knowledge state, readable by any model that can read

files. When today's tool is replaced by a better one, the files, the folder structure, and the disciplines remain; you point the new tool at the same structure and resume. That is the portability that matters, not that files import into some future proprietary format, but that plain Markdown organized into meaningful folders is a durable, human- and machine-readable representation of your working context that no vendor can lock away.

The industry is converging on the same conclusion. In June 2026 Google published the Open Knowledge Format, a proposed standard for structuring knowledge for AI agents: a directory of plain Markdown files, each carrying its metadata in a small header block, readable by humans and machines without translation. That is a description of the file layer of the vault this guide builds; the operating disciplines around it are this guide's own. The proposal is young and may change; the convergence is the point.

In practice the discipline reduces to a few habits. The operational rhythm is a load/update pair: load the relevant context files at session start, and update the running-state file at session close when the state changed, the update being the step most often skipped and the one whose skipping compounds staleness. Context sits in three layers, each doing one job, identity and universal rules at the machine level, domain and cross-project context at the vault level, project-specific context at the folder level, and what belongs at one layer is never duplicated at another. Keep to signal, not storage: the aim is the minimum structure needed for an AI, or a future you, to resume at full speed, not an archive (see Signal Over Noise). And note the transfer principle: externalizing implied context forces you to make your assumptions explicit, so the standing rules you write reveal what you actually believe about how a project should run, a benefit that is yours whether or not the AI is in the room.

Signal Over Noise

Summary: The discipline that governs the folder's context files, the files the agent loads to orient itself: each holds only what is relevant and durable at its level, and nothing else. It rules what the agent reads, not what the folder may contain.

This discipline is about the folder's **context files**, the small set of files the agent loads to orient itself, not about everything the folder holds. Three context files recur throughout this guide, and Chapter 6 builds each one properly. The first is the standing-rules file, `CLAUDE.md`, and it is special: the agent reads it automatically every time it starts in the folder, so every line in it is loaded into every session, asked for or not. (In other tools the same file is named `AGENTS.md`; Chapter 6 returns to this.) The other two are the folder's memory: a background file holding durable state and history, and, in busier folders, a working brief holding what needs attention now. The sorting rule follows: standing rules go

in `CLAUDE.md`, state and history go in the memory files, and raw notes and transcripts stay out of all of them until processed into conclusions.

Two categories must be kept separate:

- **Signal:** conclusions, decisions, positions, and frameworks that are yours and that you will act on. A synthesized vendor analysis. A finalized negotiation position. A strategic option you evaluated and rejected, with the reasoning preserved.
- **Noise:** raw input, unprocessed transcripts, verbatim meeting notes, copied articles. Noise is not waste, and it is not banished from the folder: this material has a legitimate home in the workspace as supporting resources, typically in a `resources/` subfolder. It turns into noise only when it is promoted into the context files the agent loads every session.

The discipline is not specific to any tool; it is how a well-managed knowledge system works, and the agent simply gives you the tools to enforce it and a strong incentive to, because it reads the structure in full at the start of every session. A `CLAUDE.md` that is too long becomes noise itself: a bloated file degrades focus and dilutes the instructions that actually matter, so the goal is always the smallest file that holds all the standing rules and nothing else. The same logic runs down every layer, a background file full of raw notes is doing the wrong job, and a working brief that grows into a second background file stops being scannable.

The test for what belongs where is a simple one. Before adding anything to `CLAUDE.md`, ask whether it is still true in six months regardless of what happens in this project: if yes, it belongs in `CLAUDE.md`; if there is any doubt, it belongs in `-background.md`. The same signal-and-noise split applies at every level:

Layer	Signal (keep here)	Noise (do not store here)
`CLAUDE.md`	Standing rules, always-true scope	Session history, tactical decisions
`-background.md`	Current state, decisions, history	Raw transcripts, unprocessed notes
Project folder files	Negotiation positions, decision logs, vendor assessments	Meeting verbatims, raw email threads (file them under `resources/`, not among the working files)
Folder root (intake)	A landing zone only: new artifacts arrive, get processed, then filed	Anything still sitting there weeks later

Two habits keep it honest. Curate before you execute: curation builds the vault (what should live here?), execution works from it (what can I produce from what is here?), and conflating them produces a specific failure, you execute against a noisy vault and the model amplifies the noise. And remember that every line that does not belong dilutes the lines that do, so a clean file beats a comprehensive one, and grooming stays event-driven, not session-driven, you update standing rules only when a standing rule actually changed.

Chapter 3: From Chatbots to Agents

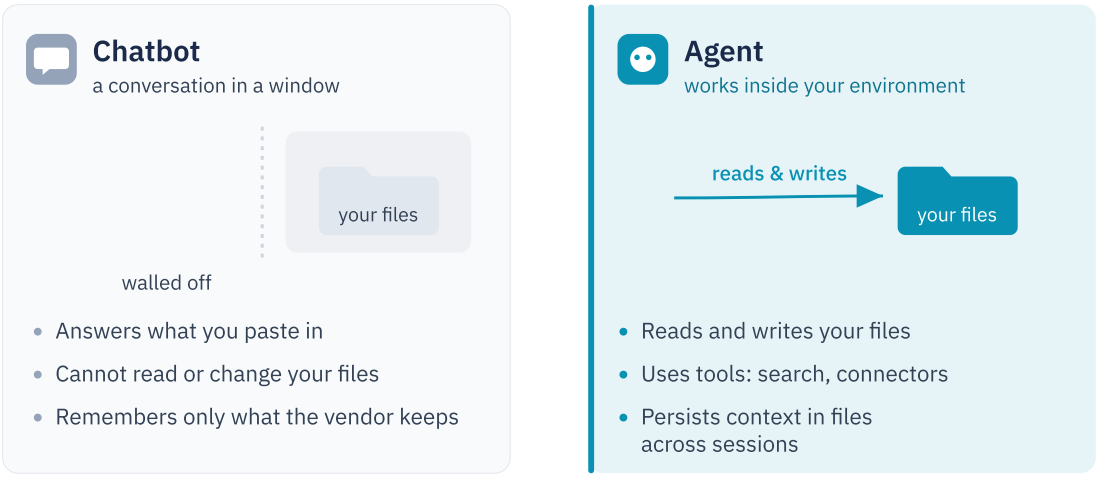
Summary: The shift from an AI chatbot to an AI agent is a change of category, not degree: an agent reads and writes your files, uses tools, and persists context, which is what makes a durable work OS possible at all. The chapter also sets the frame the rest of the guide builds on: Karpathy's three layers, the spec, the verifier, and the environment.

Where we are

Chapter 2 established the reader and the problem: high-context work, and an AI that forgets everything between sessions. Before reaching for a specific tool, it is worth being clear about the category of tool that actually solves this. The answer is not a better chatbot. It is an agent. The best-known agents of this kind, at the time of writing, are Anthropic's Claude Code and OpenAI's Codex, both installable on an ordinary Mac or Windows machine, with more arriving as the category matures, including tools built on open models. This chapter is deliberately tool-neutral; a later chapter commits to a specific one, and says plainly what carries over if you choose another.

Most people meet AI as a **chatbot**: a conversation in a browser window. You type, it answers. It is genuinely useful, and it is no longer memoryless: current chat products retain some of what you tell them and hold documents in project spaces. But the shape is fixed. It cannot see the files on your machine, and it cannot change anything in your environment. And what it does remember, it selects for you and keeps on the vendor's side, where you cannot inspect it fully, scope it to one engagement, or carry it to another tool. So you remain the integration layer: you copy context in, you copy results out, and you keep the authoritative record somewhere else, usually in your head.

An **agent** is a different kind of thing. An agent operates inside your working environment. It can read the files in a folder, write new ones, edit existing ones, and use tools, search, run a command, call a connected service, on your behalf. Crucially for the problem in Chapter 2, it can use files as durable memory: it reads standing instructions and current state at the start of a task, and it writes updated state back at the end. The conversation is no longer the only place the work lives. That is the leap. A chatbot answers questions about work you describe to it; an agent participates in the work itself, in the place where the work actually sits.



A chatbot works from what you hand it, and what it retains stays with the vendor; an agent reads and writes your files, uses tools, and persists context you own.

Two capabilities make the agent the right foundation for a personal work OS, and both are absent from a chatbot. The first is **agency over your environment**: because the agent reads and writes real files and uses real tools, the output is not a block of text you then file, format, and apply by hand, the agent acts on the workspace directly, and the copy-paste tax disappears. The second is **persistence through files**: because context is stored as files the agent reads on the next run, the work compounds, and what you established last week is on disk, yours to read, correct, and carry between tools, not held for you inside someone else's product. This is the mechanism that answers the stateless problem, the model still has no memory, but the system around it does, and you own the system.

This is also the direct rebuttal to "I thought that was only for coding." Agents grew up in software development because code lives in files and developers wanted AI that could act on those files. But nothing about reading folders, writing documents, persisting state, and using tools is specific to code. A negotiation folder, a board-deck folder, and a vendor-research folder are exactly the same shape: files in a directory, context that must persist, work that pauses and resumes. The capability that makes agents valuable to developers is the same one that makes them valuable to the high-context professional.

So the decision that matters is the category, not the product: chatbot versus agent first, which agent second. The file is the unit of memory, an agent's durability comes from reading and writing files, not from any built-in memory feature, so designing around files keeps you portable across whichever agent you use (see Capturing and Supplying Context in Chapter 2). Agency is the end of copy-paste, the more of your real work surface the agent can touch directly, the less you broker between the AI and your work. And "only for coding" stays a

framing error, because the agent does not care whether a file holds code or a commercial position; it reads, writes, and acts the same way on both.

The Three Layers of Working Well

The case so far is for the category: an agent over a chatbot. Before the guide commits to a particular agent, it helps to have a picture of what doing this *well* looks like, because everything that follows is an extended answer to that. The clearest picture comes from Andrej Karpathy, who led AI at Tesla and has spent more time than almost anyone watching these models behave.

He starts with a small test. Ask any current model: I want to go to a car wash 50 meters away, should I drive or walk? Every model says walk, because 50 meters is close. They all miss that you cannot wash a car you did not bring. The point is not that the models are weak. They are strong on anything that can be measured and blind on anything that turns on context they were never given. The whole method exists to close that gap, and so does this guide.

Karpathy breaks effective work into three layers that stack, and they are worth naming now, because each chapter ahead builds one of them.

- **The spec** is how you hand your understanding to the model before it builds anything. You uncover the goal instead of stating the task, you work in small pieces with checkpoints instead of handing over the whole job at once, and you make the model confirm its key assumptions before it commits. The guide builds this in folder-init, where the agent interviews you to draw out what only you know, and in plan mode, where it shows its approach before touching a file.
- **The verifier** is how you check what comes back against a standard you set, instead of trusting it because it sounds sure. Karpathy's image for why this is needed is a robot librarian: it answers only from the books in its library, it does not know which books it is missing, so when it lacks the right one it invents an answer with full confidence. Pleading with it, or telling it to "make it better," does nothing, because those are levers for a person, not a librarian. The only real lever is verification: state the criteria up front, have a second pass grade the first, and connect the work to the place the truth actually lives. The guide builds this in the review gate and in the librarian's discipline of tracing every claim back to a source. (Chapter 12 returns to why this layer works least like it does for engineers, and what verification becomes when the work is knowledge, not code.)
- **The environment** is the workspace the spec and the verifier live in, built once and reused, so your context, your rules, and your tools are waiting every time instead of

being re-explained from scratch. A standing-rules file the agent reads on every run. A knowledge base of your own data, because, in his words, your data is your moat. A library of skills you sharpen by using them. And rules sorted into always-do, ask-first, and never-do, with the never-do ones enforced rather than merely asked for. The guide builds the environment across most of its length: the CLAUDE.md system, the folder, the memory layer, and the skills.

His closing line is the one to keep. *You can outsource your thinking, but you cannot outsource your understanding.* The thinking is the labor the model is glad to take. The understanding, what the work is for and which detail is load-bearing, stays yours, and the three layers are how you move it into a form the model can act on.

One word of caution, because this guide uses “layer” for other things. These three are not the machine, vault, and folder layers you meet in Chapter 6, nor the maturity ladder of Chapter 12. Those describe how your files nest and the order you build them in. Karpathy’s three describe the things you *do* to get good work out of an agent. Keep them apart in your head.

Here is the part worth sitting with. None of this was the plan for the system in this guide. The system was built from practice, one folder at a time, before the frame had a name. An independent and credible source arrived at the same architecture from the other direction. So as you read on, you are not being handed a theory. You are building the spec, the verifier, and the environment, and watching them line up with a frame reached separately. That convergence is the best evidence the shape is right.

Chapter 4: One Estate, Two Lenses

Summary: The work OS is one estate, the vault, its folders, and the context files, seen through two lenses: what it remembers and what it does. The “second brain” the internet sells is one room in that estate, not the estate itself. Getting the words straight here is what stops you building the wrong system.

Where we are

The last chapter made the case for agents. Before committing to a tool, it is worth clearing the words, because the words are where most people get lost. The names for this kind of system are a fog: “second brain,” “AI OS,” “agentic OS,” “personal knowledge management,” “[your name]OS,” “work OS.” Each influential voice online bends the same handful of terms to mean something slightly different, and following one voice is never enough. Follow several and you end up where many capable people end up, including the author for a while: unsure whether these are one thing or many, and unsure which one you are supposed to be building.

This chapter settles the words. It is short on purpose.

One estate, two lenses

Start with the claim and then defend it. You are building **one estate**: the vault, the folders inside it, and the plain context files the agent reads. Not two systems. One.

You look at that one estate through two lenses:

- **What it remembers.** The knowledge layer: your standing rules, background files, working briefs, transcripts, the memory of every decision. This is the part that holds still and waits to be read.
- **What it does.** The action layer: the skills, the automations, the scheduled routines, the agent reaching into a folder and producing something. This is the part that moves.

A useful framing from the field is “two layers, one system.” The knowledge layer comes first; the action layer is built on top of it, and is worth nothing without it. An automation built on thin context acts confidently and wrongly. That is the whole dependency: you cannot have a system that acts well until you have a system that remembers well.

The reason this is one estate and not two is that the same files serve both lenses at once. A `CLAUDE.md` is knowledge, the agent reads it to know who you are and how this folder works, and it is also routing, it tells the agent where to go and what to do. A file that both remembers and directs cannot be cleanly assigned to one layer. The layers are real; the wall between them is drawn for convenience, and it moves depending on what you are doing.

The second brain is a room, not the house

The most loaded word in the fog is "second brain." When most people say it, they mean what Amer built in Chapter 1: a growing, cross-linked library of their own knowledge, fed by everything they read and watch. That is real, and this guide builds it later (see Building a Knowledge Base Wiki in Chapter 11). But it is **one room in the estate, not the estate itself.**

This matters because the popular content collapses the two. It shows you a "second brain" and implies that building it is the same as building your work OS. It is not. Amer's knowledge library accumulates expertise and surfaces it as a wiki. Ken's contract folder and Lama's negotiations run live work and close it out. They are different kinds of room, with different jobs, different failure modes, and different upkeep. The guide keeps them apart on purpose, and the fork between them is covered in full in Work OS vs Knowledge Base.

So when you hear "second brain," translate it: *a knowledge-accumulation use case, one application of the work OS.* It is a room. You are building the house.

Why the seam is still useful

If it is all one estate, why keep the two lenses at all? Because the lens tells you which discipline to apply.

- The **knowledge layer** fails by going stale, contradicting itself, or losing its source. You guard it with freshness checks and the occasional lint.
- The **action layer** fails by doing the wrong thing, at the wrong time, or with no one watching. You guard it by making each automation earn its keep and proving it is watched.

Knowledge is safe to accumulate; you rarely regret keeping a note. Automations are not safe to accumulate; an unwatched routine with a side effect is a liability. Asking of any file, "is this here to be remembered, or to be acted on?" tells you which of these you are dealing with. The seam is a working distinction inside one estate, not a wall between two.

A short lineage

It helps to see where the words came from, because each generation added a layer and kept the old name.

- **The second brain** (Tiago Forte) was a system to capture and retrieve your own knowledge. The human was the actor; the brain just remembered.
- **The LLM wiki** (the pattern behind Amer's folder) handed the upkeep to an agent: now the system grows and cross-links itself, but it still mostly remembers.
- **The agentic OS** folds an actor into the same system: now it remembers and acts.

At each step the field kept the words and appended to them, which is why "second brain" and "AI OS" and "agentic OS" now overlap so badly. The guide's position is simple: when the actor arrives, you do not buy a second system. You grow a second layer on the one estate you already keep.

A full crosswalk of the terms you will meet online, what each usually means, who uses it, and where it lands in this guide's vocabulary, is in Appendix D. Reach for it whenever a video leaves you unsure which thing is being sold.

Key Principles

1. **One estate, two lenses.** The vault, its folders, and its context files are one system. "Remembers" and "acts" are two ways of looking at it, not two things to own.
2. **The knowledge layer comes first.** The action layer is built on it and is worthless without it. Build the memory before you automate.
3. **The second brain is a room.** It is the knowledge-accumulation use case, one application of the work OS, not the work OS itself. Do not mistake the room for the house.
4. **Let the lens pick the discipline.** Ask whether a file is to be remembered or acted on; the answer tells you whether to guard it with freshness or with restraint.
5. **When the words confuse you, go to the crosswalk.** Appendix D maps the popular terms to this guide's vocabulary.

Chapter 5: Getting Set Up

Summary: An operating system starts with a habit, not an architecture: route everything through one agent and let the context accumulate. This chapter covers the one early choice you do need to make, where to run the agent, and a small input habit, speech-to-text, that changes how the tool feels to use.

Where we are

You have the problem, the agent, and the words. Before the files and folders, two practical things: the habit that makes the whole system work, and the place you will run it from. Neither needs much. The granular setup lives in Appendix B; this chapter is about getting started, not about configuration.

The default before the architecture

An operating system does not start with architecture. It starts with a default.

The mistake is to think you must design the whole vault before you can begin. You do not. The first move is a habit: when there is a task in front of you, route it through the agent, in a folder, instead of scattering it across a browser tab here, a chat window there, a custom assistant somewhere else. Every time you repeat yourself to a fresh chat, you are paying a tax the work OS exists to remove. The fix is to make one agent, in one place, your default destination for work.

The structure follows from the habit, not the other way around. Once everything runs through one channel, context starts to accumulate in one place, and the folders, the rules, and the files assemble themselves around the work that actually happens. Build the habit first. Let the architecture catch up. The whole of Chapter 12 is this principle restated: the system matures through use, it is not designed up front.

Before You Start: Account, Install, and a First Check

Three facts to have straight before anything is installed. First, the account: Claude Code runs on a Claude account, and recurring use of the kind this guide describes sits on a paid plan. The current tiers and prices are on the vendor’s pricing page, and the same goes for Codex and its OpenAI account; expect a monthly subscription in the range of other professional

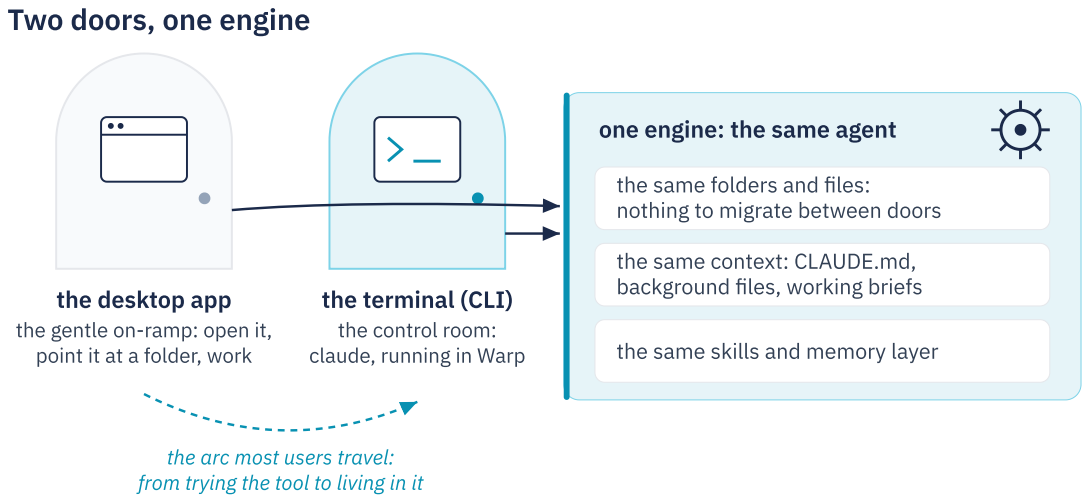
software, and check the live numbers rather than trusting any book to stay right about prices. Second, the install: this guide deliberately prints no install commands, because they change faster than any book can. The vendor's own install page carries the current steps for both the desktop app and the CLI, and following it takes minutes on a machine you control; on a managed corporate machine, expect to need IT approval first, the same pause Appendix B describes for connectors. Third, the check that it all worked, run once you have written your first standing-rules file in Chapter 6: open a session in that folder and ask, "what standing rules are you operating under?" If the agent answers from your file, the system is live, and context is loading before your first word, which is the whole point of the chapters ahead.

Where to run the agent

Claude Code, the agent this guide commits to in the next chapter, runs in more than one place, and the differences are about comfort, not capability. The engine and the files are identical wherever you run it. Four common choices, with who each fits:

Environment	What it is	Who it fits
Native terminal	The terminal built into macOS or Windows, running `claude`	Anyone; the lowest-setup starting point, plainest interface
Claude desktop app (Code)	The Claude app's own working surface	A non-technical start, though it still leans on the terminal underneath
VS Code extension	The same terminal session, docked inside the developer editor	Developers, who get panes for code and diffs beside the agent; richer than most readers need
Warp	A modern terminal with an editor-like interface (iTerm2 is the other widely used terminal upgrade)	The recommendation here: terminal power with a gentler surface

There is a natural arc here, and it is worth naming because most people travel it. You start in the **desktop app**, because it is the gentlest on-ramp: you open it, point it at a folder, and work, with no terminal in sight. As your work OS deepens, more folders, the odd script, a plugin to install, a new model to switch to the day it ships, you start wanting the control the terminal gives, and you graduate to the **CLI**, running `claude` in a terminal. This is the same migration the next chapter calls "two doors, one engine": the engine never changes, only the door you prefer, as you move from trying the tool to living in it. Most maturing users end up in the terminal, and the terminal worth ending up in is Warp.



Two doors, one engine: the desktop app and the terminal open into the same agent and the same files; maturing users drift from the first door to the second.

The honest account, from building this system in practice: the starting point was the desktop app (Code), which still required the native terminal underneath. The first real software project pushed the work into VS Code, where the agent runs as the same terminal session inside the editor; it turned out to be built for developers and heavier than knowledge work needs. That led to Warp, and Warp is where it settled. Warp gives a far richer terminal, an interface with the comfort of an editor, a folder explorer, and multiple tabs, without the developer weight of VS Code. It also has its own built-in AI assistant in the shell: you describe in plain words what you want the terminal to do, and it suggests the command.

One small story shows why that matters for this reader. The day a new Claude model shipped, it did not appear in the list of available models when starting a Claude Code session. Rather than hunt through documentation, the question went to Warp's AI assistant in plain English: how do I update the Claude Code plugin? It returned the commands, which ran, and the new model appeared. Up and running on the new model, hours after release, with no technical research. That is the experience to aim for: the environment closes the gap between what you want and the command that does it.

The recommendation: start in whatever is in front of you, the native terminal or the desktop app, and move to Warp once you are working daily. It is the best fit for a non-technical reader who wants terminal capability without a developer's tooling. One honesty note on platform: the environment described here is macOS-tested, and so are the setup procedures in the appendices. On Windows the same agents and the native terminal work, but the surrounding recommendations, dictation above all, have macOS-specific steps you will need to translate.

Speech-to-text: an input habit, not a context source

One habit changes how the tool feels: dictating instead of typing. It belongs here, not in the chapters on context, because it is about how you operate the agent, not about what the agent knows.

The honest account again: the first try was Whisper, and the experience was slow and unintuitive. Then came Spokenly, which fixed those complaints and works almost without thinking, inside Warp and inside any other macOS application. SuperWhisper was not tried directly, but enough users online report an experience close to Spokenly's that it is worth naming as the alternative. Either one turns a long prompt from a typing chore into a spoken sentence, which lowers the cost of giving the agent the full context every time, the thing this guide keeps asking you to do.

Granular setup for both the environment and speech-to-text is in Appendix B. Do not over-invest here. The point of this chapter is to start, not to perfect.

Key Principles

1. **Default first, architecture later.** Route work through one agent in one place before you design any folder structure. The structure follows the habit.
2. **The environment is comfort, not capability, and it evolves.** The same engine and files run everywhere, so pick the surface you are happiest in. Most readers start in the desktop app and graduate to the CLI in a terminal as their work deepens; Warp is the terminal to land on.
3. **Lower the cost of giving context.** Dictation (Spokenly or SuperWhisper) makes full briefing cheap, which is what the rest of the guide depends on.
4. **Start with what is in front of you.** Do not let setup become the project. Begin, then improve only when friction appears.

Chapter 6: Claude Code: The Agent of Choice

Claude Code Fundamentals

Summary: Claude Code is the agent this guide commits to: full read/write access to your local files, persistent context through plain Markdown, available as a desktop app for non-technical use and a CLI for developers, both reading the same files.

Where we are

Chapter 3 made the case for agents in general; the two chapters since cleared the words and got you set up. This chapter gets concrete. From here on the guide commits to one agent, Claude Code, because it reads context files natively, runs inside an ordinary folder of Markdown, and respects a folder hierarchy without configuration. Everything in this chapter and the three that follow is specific to it. The portability argument still holds underneath: the files you build outlast the tool. From here through Chapter 9, the standing-rules file, the folder, and the memory layer, you are building the environment layer from Chapter 3: the workspace the agent works inside, set up once and reused.

One note before committing, for the reader whose agent is not Claude Code. Everything from here on transfers directly to OpenAI's Codex and to the other agents of the category, with one mechanical difference worth knowing: the standing-rules file this guide calls `CLAUDE.md` is, in Codex and a growing list of other tools, a file named `AGENTS.md`, an open standard doing the same job. Read `CLAUDE.md` throughout as "the standing-rules file your agent reads automatically," and the rest, the folders, the background files, the disciplines, applies unchanged.

Claude is available in several modes, and the difference that matters is not power. It is **persistence**: not whether anything is remembered, but where the working context lives, what it covers, and whether you control it.

Mode	What it does	Key limitation
Claude.ai (Chat)	Conversational Q&A, one-off tasks	Memory is product-managed and vendor-held; no access to your local files
Claude.ai (Projects)	Attach documents as standing background	Read-only: can reference your files but cannot modify them
Claude Code (desktop app or CLI)	Full read/write to local files, persistent context via `CLAUDE.md`, full session control	Requires selecting a folder and a one-time `CLAUDE.md` setup

For substantive, recurring knowledge work, Claude Code is the agent of choice. The others are for quick, isolated lookups. Only Claude Code can read your files, update them as you iterate, and carry standing context across every future session for that folder.

Once a `CLAUDE.md` file exists in a project folder, every session starts with the agent already briefed: your role, priorities, operating rules, and tone preferences. You do not re-introduce yourself or re-state format requirements. You open the app, select the folder, and start working. That is the core of the transition; everything in the next chapters builds on it.

Key Principles

1. **Setup is three steps.** Open a session pointed at your project folder, from the desktop app or by typing `claude` in a terminal there, and your context loads automatically: if a `CLAUDE.md` exists in that folder or any parent, Claude Code reads it before your first word.
2. **Two doors, one engine.** Claude Code has a desktop app and a CLI accessed by typing `claude` in a terminal. The engine, the files, and the conventions are identical: `CLAUDE.md`, `-background.md`, slash commands, the `@file` syntax, the directory-tree walk at session start. Most online content shows the CLI because it is older and the default for technical users. Read past the interface details; the files and conventions transfer cleanly. The desktop app is the gentler starting door; most maturing users end up in the terminal, as Chapter 5 described, and both doors share the same `~/.claude/` directory, so nothing is lost in the move.

3. **Markdown is the format behind everything.** `CLAUDE.md`, `-background.md`, and reference files are all plain-text Markdown. A `#` is a heading, a `-` is a bullet, `**word**` is bold. Markdown is the standard across AI tools, carries no proprietary lock-in, and makes every file you write a portable knowledge asset. You do not need to learn the syntax first; the agent drafts the files from plain-language instructions.
4. **Persistence, not intelligence, is the upgrade.** The model is the same across modes. What changes is that the folder remembers, so the next session is productive from the first message.

The CLAUDE.md System

Summary: `CLAUDE.md` is the standing-rules file the agent reads automatically at every session start, and it lives at three nested layers, machine, vault, and folder, each doing one job.

Core Concept

A `CLAUDE.md` is a Markdown file at the root of a project folder. Claude Code reads it on launch without being asked. It is the single most important file in the vault, and it holds standing rules only.

What belongs in it:

- Your role and organization
- The purpose and scope of this specific folder
- Tone and format rules (numbered points, no marketing language, 24-hour time)
- Decision frameworks you consistently apply (for example, Situation / Key Insight / Options / Recommendation)
- Sensitivities and out-of-scope topics

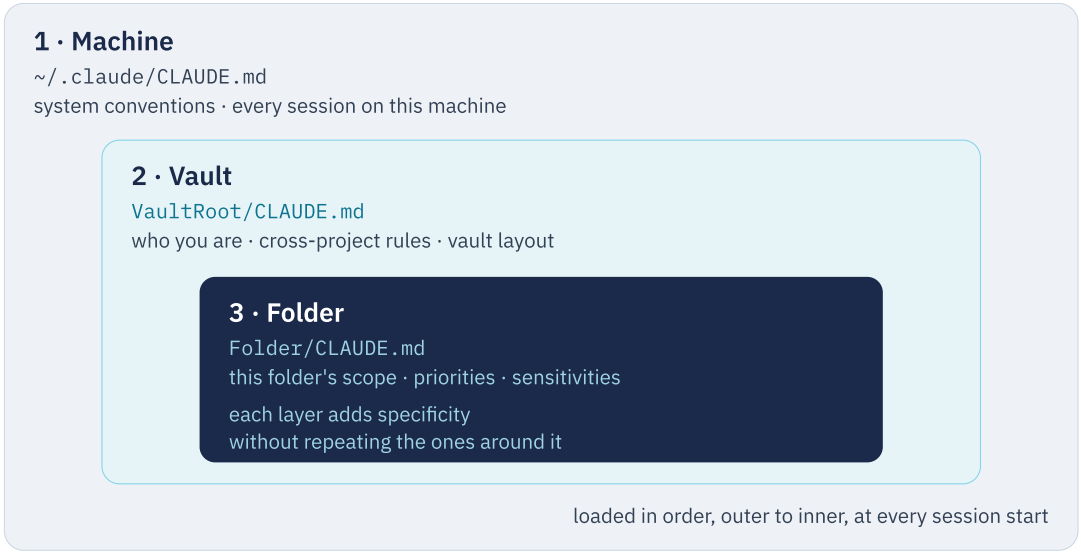
What does not belong:

- Specific meeting outcomes or task history
- Current negotiation positions or timelines that change frequently
- Session notes or anything time-bound
- Anything you would not want applied to every future session in this folder

The Three Layers

When Claude Code launches from a project folder, it walks up the directory tree and loads every `CLAUDE.md` it finds, plus a special file at `~/.claude/CLAUDE.md`. Three layers, in load order, each scoped differently:

Layer	File	Scope	Holds
Machine	`~/.claude/CLAUDE.md`	Every session on this machine, any folder	System conventions: file-structure SOPs, naming rules, output-style defaults
Vault	`VaultRoot/CLAUDE.md`	Every session inside this vault	Identity and cross-project context: who you are, organizations, vault layout, operating rules
Folder	`Folder/CLAUDE.md`	Sessions in this specific folder	Project-specific scope, priorities, tone for a counterparty, what is out of scope



The three CLAUDE.md layers nest from machine to vault to folder, each loaded in order at session start.

Each layer does one job. You never repeat content across layers. If a rule is being copy-pasted into multiple files at one level, it belongs one level up. A quick test with three real rules: “always use 24-hour time, and never open an email with pleasantries” is true of everything you write, so it lives in the machine layer; “these are my organizations and the people I work with” is true across projects, so it lives at the vault; “we are mid-negotiation with this vendor and the price ceiling is confidential” is true only here, so it lives in the folder. Put each rule where it is true and you never write it twice. The machine layer keeps your structural discipline portable across every project on the machine, including projects that are not inside the vault at all.

In a mature vault the vault layer does real work. The vault-root `CLAUDE.md` is where you tell the agent, once, to read the cross-project background, the terminology invariants, and the entity intelligence index at the start of every session, so every folder beneath it inherits that orientation. The Folder as the Organizing Artifact shows that inheritance in action.

Why It Matters

Without `CLAUDE.md`, every session starts cold and you compensate with long, inconsistent prompts. With it, your context is always loaded before you type. Without the machine layer specifically, the same structural conventions get duplicated into every vault root file or drift between vaults.

Key Principles

1. **Let the agent draft the first file.** Setting up a new folder is the one moment a blank page is genuinely useful. Describe the folder's purpose and the standing rules you want, and ask for a draft `CLAUDE.md` and `-background.md`, applying signal/noise. You supply the substance; the agent supplies the structure. In a mature vault this is what the folder-init skill automates.
2. **The machine layer is machine-local.** `~/.claude/CLAUDE.md` is not carried by vault sync. Across machines, either sync `~/.claude/` via a dotfiles repository or maintain a parallel copy and reconcile. The same caveat applies to auto-memory.
3. **Apply the six-month test.** Anything still true in six months belongs in `CLAUDE.md`; anything that changes belongs in the background file. See Signal Over Noise in Chapter 2.
4. **Groom event-driven, not session-driven.** Update `CLAUDE.md` only when scope or a standing rule materially changed, never just because a session happened. A clean file beats a comprehensive one. Review all folder `CLAUDE.md` files quarterly.
5. **Context isolation by folder is intentional.** A session launched from one folder does not carry another folder's context. Keep each `CLAUDE.md` scoped strictly to its own folder; use `/add-dir` only when a task explicitly needs cross-folder context.

The Background File

Summary: The `-background.md` file holds a folder's running state, and it works through a load/update pair, an automatic read at session start and a deliberate write at session close, where the write side is the part that quietly fails.

Core Concept

Alongside each folder's `CLAUDE.md`, a companion file named `[FolderName]-background.md` holds the specifics: content that matters now but changes over time.

What goes in it:

- Negotiation history and current positions
- Key decisions made and their rationale
- Stakeholder dynamics and political context
- Outstanding items and open questions
- Context important today but possibly not in six months

The background file has no native auto-load. Unlike `CLAUDE.md`, it loads only when something tells the agent to read it. The recommended approach is a standing instruction inside `CLAUDE.md`:

```
### Session Start
At the start of every session in this folder, read FolderName-background.md
before responding.
```

The agent follows this natively, so the file loads automatically every session without you remembering to pull it in.

A convention worth adopting early: give each section a dated header, `## YYYY-MM-DD: Section Title`, where the date means "last updated." New content gets a new dated section; changed content keeps its section and moves the date forward. This keeps the file a coherent narrative rather than an append-only log, and it is exactly the format the session-handoff skill maintains automatically.

The Load/Update Pair

The mechanic that makes a memory file actually work is a pair of intentional actions: a read on session start and a write on session close.

The read side automates cleanly. Three options of increasing automation: a manual `@FolderName-background.md` reference, the eager-load instruction above (the default for active folders), or settings-level automation via hooks (rarely worth the complexity).

The write side does not automate cleanly, and this is where the system quietly fails. People believe their memory file is updating because the setup *feels* persistent, but it only updates when something explicitly tells it to. Six months in, the background file reflects January and the project has moved on twice. The fix is the discipline of pairing every intentional read with an intentional write. The simplest write is an explicit closing prompt:

```
Update FolderName-background.md with any decisions, position changes,
or context shifts from this session that should persist. Keep it
structured: decisions, open items, current state.
```

Reviewing that update before closing is the single most important habit in the whole workflow. Chapter 8 shows how the session-handoff skill turns this close into one

command, and treats separately the question of how far beyond the folder a close should ever reach, which is a design choice, not a default.

The Asymmetry Worth Naming

You can almost always automate "read this." You cannot fully automate "remember this well." Updating a memory file requires the agent to summarize what happened, decide what is worth keeping, and integrate it without losing prior content. That is content judgment, not a load instruction, which is why a closing prompt or a skill (both of which carry instructions about *how* to update) outperforms a hook (which only triggers *when*). A human-in-the-loop closing prompt is not a workaround; it is the right shape for the task.

Eager vs Lazy References

`CLAUDE.md` is also the routing layer pointing to other files, with two kinds of pointer:

1. **Eager references (always loaded).** Files the agent must read every session, declared with a `Session Start` instruction. The background file is the classic case. Use sparingly: every eager file costs context window in every session.
2. **Lazy references (loaded on demand).** Voice profiles, frameworks, templates, archived analyses. Declared as a routing list with a one-line trigger for each. The agent reads the cheap routing list and opens the actual file only when the task matches. A useful convention is to keep these in a `resources/` subfolder.

Key Principles

1. **Update after material change, not after every session.** Keep the file structured (decisions, open items, current state) so it stays readable.
2. **The naming is convention, not standard.** Online content often calls the running-state file `MEMORY.md`; it is the same concept. Only `CLAUDE.md` is native to Claude Code. Pick one name and stay consistent.
3. **Do not confuse it with the agent's auto-memory.** A separate store, also indexed by a file called `MEMORY.md`, lives under `~/.claude/` and is written by the agent. See Auto-Memory in Chapter 9.
4. **Graduate the write side deliberately.** Start with the manual closing prompt. Move to the session-handoff skill once you know what a good close looks like. Use a Stop hook only at high session volume.

The Working Brief

Summary: When a folder shifts from knowledge storage to active operations, a third file, `[FolderName]-workingBrief.md`, separates durable memory from current operational state; the background file is the memory, the working brief is the steering wheel.

Core Concept

The two-file structure (`CLAUDE.md` plus `-background.md`) is the right default for most folders. It works whenever the primary activity is knowledge work: collecting context, developing thinking, managing a bounded topic, preparing occasional deliverables.

Some folders shift character. When knowledge work starts feeding decisions, coordinating inputs from multiple people, and running parallel delivery tracks, the folder has become operational. The strain shows in the background file, which ends up doing two jobs at once: holding settled framing alongside tracking notes for pending input, a flag that a team has not delivered numbers, an open question about whether an ask is confirmed.

The settled framing answers "what do I need to know to work on this intelligently?" and does not change between sessions. The tracking content answers "what needs attention this week?" and changes every session. When a background file starts mixing the two, it is time to separate them.

File	Question it answers	Changes when
`CLAUDE.md`	How should the agent behave in this folder?	Standing rules change
`[FolderName]-background.md`	What must I know to understand this work?	Something becomes part of the lasting record
`[FolderName]-workingBrief.md`	What needs attention right now?	Priorities, open questions, or workstreams shift

Why It Matters

The background file is the memory. The working brief is the steering wheel. Neither replaces the other. Without the split, an operational folder's background file becomes too heavy to re-orient from quickly, and the current state gets buried under stable context.

When to Add a Working Brief

Good signals:

- Multiple inputs arriving from different people or teams across sessions
- Parallel workstreams each with their own current state
- Open decisions actively blocking next moves
- Strategy, communication, and execution happening simultaneously
- Work that pauses and resumes across sessions or machines where the background file alone is too heavy to re-orient from

When not to:

- A single-person deliverable with no coordination overhead
- A reference or dormant folder where the background file handles re-orientation
- Any folder where a third file creates maintenance rather than reducing it

The working brief earns its place only when it reduces the cognitive load of returning to a folder. If it starts to feel like paperwork, the folder does not need it. This is the incremental design principle applied to file structure. In a mature vault both the folder-init and session-handoff skills apply exactly these criteria when deciding whether to create one.

Key Principles

1. **Add a second eager-load instruction.** Update `CLAUDE.md` so both files load at session start: "read [FolderName]-background.md and [FolderName]-workingBrief.md before responding."
2. **Session-close routing for three-file folders.** For each item: does it answer "what became part of the lasting story?" (background file) or "what needs attention now?" (working brief)? If both, write it in different forms: the settled decision to the background file, the open action that follows to the working brief.
3. **The working brief is full-replace and always current.** Unlike the background file, which you update when something becomes permanent, the working brief should always reflect the present. A working brief two sessions out of date is worse than none: it misdirects the next session rather than orienting it. Carry a `Last updated: YYYY-MM-DD HH:MM` line.
4. **Record the previous working-brief state before overwriting.** Capture it as a dated summary in the background file first, so the steering history is preserved in the memory layer. The session-handoff skill does this automatically.

5. **Keep it short and scannable.** Its value is in being current, not comprehensive. Comprehensive is the background file's job.

A Working Brief Shape

```
## Working Brief: [Folder Name]
Last updated: YYYY-MM-DD HH:MM

### Current Objective
### Active Workstreams
### Current Decisions
### Open Questions
### Next Moves
### Current Risks or Sensitivities
### Most Relevant Artifacts
```

How Standing Rules Fire: Reflex Rules and Hooks

Summary: Two different mechanisms make the system act on its own. A reflex rule is an instruction in `CLAUDE.md` the agent applies with judgment, and it is how a skill "auto-triggers." A hook is a command the harness runs on a mechanical event, deterministically, and it is how you enforce a rule that must never be broken. Knowing which is which is most of what it takes to automate well.

Core Concept

The three files this chapter built answer what the agent should know. This section answers how the rules inside them actually fire. The load instruction you wrote into `CLAUDE.md`, read the background file before responding, happens every session with nothing forcing it; a rule that must never break, such as protecting a file from being overwritten, needs more force than a sentence the agent is asked to honor. Two separate mechanisms cover that range, and when people want the agent to "do something automatically," they usually reach for the wrong one.

A **reflex rule** is a standing instruction in `CLAUDE.md`. The agent reads it on every prompt and applies it with judgment. "At the start of every session, read the background file" is a reflex rule. So is "when you are about to state a claim you cannot source, find the source first." This is how a skill appears to fire on its own: there is no machinery behind it, the agent

simply notices the moment the rule describes and acts. Reflex rules are for triggers that require understanding: deciding that a claim is unsupported, that a discussion has turned to something worth recalling, that an artifact belongs in another folder.

A **hook** is a shell command the harness runs when a fixed event fires: a session starts, a prompt is submitted, a tool is about to run, a session ends. It runs whether or not the agent would have chosen to, and it is blind to meaning, it knows a file is about to be written, not what the writing is for. That blindness is the point. It makes hooks right for two jobs: enforcement (a `PreToolUse` hook can inspect a file path and refuse the write) and capture (a hook can record every exchange, or inject the same context at every session start, with no judgment needed). MemSearch, in the next chapter, uses one of each: capturing every turn is a mechanical event, so it is a hook; deciding that the current question would benefit from past context is a judgment, so recall is a reflex rule.

The dividing line

Ask one question of anything you want to happen automatically: is the trigger a mechanical event, or a judgment about meaning?

- A mechanical event (a tool runs, a session opens) is a **hook**.
- A judgment about meaning (this needs a source, this is worth recalling, this should be checked) is a **reflex rule**.

Get it wrong in either direction and you pay for it. Try to express a meaning-based trigger as a hook and it cannot work; assume judgment-based automation is impossible without a hook and you never write the rule that would have delivered it.

Why It Matters

This is also where the environment layer's guardrails live. Karpathy's frame sorts every protected action into always-do, ask-first, and never-do. `CLAUDE.md` handles the first two well: standing rules the agent reads and applies. But a line in `CLAUDE.md` is a request, and the agent can step over it. For the few rules that must never break, do not protect a canonical file or an immutable `raw/` folder with a sentence the model is asked to honor; protect it with a hook that denies the write at the tool level. Same intent, two very different levels of force. The rule you only wrote is a request. The rule you enforced with a hook is a wall.

Key Principles

1. **Mechanical event, use a hook. Judgment about meaning, write a reflex rule.** This one question resolves almost every "how do I automate this" decision.
2. **Skills auto-trigger through reflex rules, not hooks.** The standing instruction in `CLAUDE.md` is what makes the agent reach for a skill at the right moment.
3. **Enforce never-do rules with hooks.** A `CLAUDE.md` line is a request; only a hook makes a rule the model cannot break.

Where a Rule Lives: CLAUDE.md or the Skill

Summary: Every reader who builds a second skill hits the same question: a new rule exists, so does it go in `CLAUDE.md` or in the skill? One test answers it: if the rule must fire before the agent knows it needs the skill, it is a trigger and belongs in `CLAUDE.md`; if it only executes inside the procedure, it belongs in the skill. The economics enforce the test: every `CLAUDE.md` line is paid for on every turn of every session, while a skill's body costs nothing until it fires.

Core Concept

The previous section explained how a standing rule fires. This one answers where a rule must live to fire at all, and the question arrives the moment there are two places a rule could go. A skill is a named procedure the agent loads only when it is invoked; Chapter 8 introduces the two this system grows first, `folder-init` and `session-handoff`, and Chapter 12 names the moment a routine earns promotion into one. Once a skill exists, every new rule poses a choice: into `CLAUDE.md`, or into the skill?

Both wrong answers are common, and they fail differently. Rules pile into `CLAUDE.md` because it is the file you already know, until it is a three-hundred-line document degrading every session that loads it. Or rules hide inside skills that never fire, and you conclude the rule "did not work." Both mistakes treat the two files as interchangeable places to put a sentence. They are not. One test separates them:

> Does the rule have to fire before the agent knows it needs the skill? If yes, it belongs in `CLAUDE.md`. If it only executes inside the procedure, it belongs in the skill.

`CLAUDE.md` holds triggers, invariants, and prohibitions. The skill holds procedure. Put another way: `CLAUDE.md`'s job is to know that a door exists and when to open it. What is

behind the door is the skill's business.

Worked on a real change

One upgrade to a folder-creation routine produced seven rules in a single session. Before the test, all seven felt like "rules about creating folders" and could have gone anywhere. They split cleanly:

Rule	Where it went	Why
A folder with no context files gets initialized before any work happens in it	`CLAUDE.md`	It must fire on a session that has not yet decided it needs the skill. It is the trigger; nothing runs without it.
Deciding create-or-extend is a root-session job when genuinely in doubt	`CLAUDE.md`	It fires when no folder exists yet, so no folder-level file can carry it.
Substantive work never happens at the vault root	`CLAUDE.md`	An invariant. It constrains the session itself, not any procedure.
Check the vault map for an existing folder covering the same scope	Skill	A step inside the procedure. Costs nothing until a folder is being initialized.
Challenge the folder's name against its siblings	Skill	Only meaningful once a folder is being built.
Rename now, while it is still one cheap move	Skill	Same.
Search the mailbox for the counterparty's correspondence	Skill	Same.

Three rules in `CLAUDE.md`. Four in the skill.

Why It Matters

The economics make this a real decision, not a matter of taste. Every line in `CLAUDE.md` is paid for on every turn of every session, forever. A skill's body is free until it fires. That asymmetry is the whole argument: the standing-rules file is the most expensive space in the system and the only file guaranteed to load, so it must hold exactly what cannot be discovered on demand, and nothing else. A skill can be long, detailed, and checklist-shaped at no standing cost, which is exactly where procedure wants to live. This is the discipline Signal Over Noise (Chapter 2) applies to context files, stated one level up: the skill is a reference file, and `CLAUDE.md` keeps only the pointer and the trigger.

One corollary, learned by catching the mistake mid-edit: a trigger rule in `CLAUDE.md` must not enumerate what the skill does. A helpful list of the skill's steps duplicates facts that live in the skill and goes stale the moment the skill changes. One clause is enough: the skill owns what happens next; this rule only says it must happen.

The choice matters beyond housekeeping. A rule in the wrong file is documentation; a rule in the right file is behavior. The previous section drew the line between a request and a wall. This is the line before that one: a rule that never loads is not even a request.

Key Principles

1. **Triggers, invariants, and prohibitions live in `CLAUDE.md`; procedure lives in the skill.** If the rule must fire before the agent knows it needs the skill, it cannot live inside the skill.
2. **`CLAUDE.md` is the most expensive space in the system.** Every line is paid for on every turn of every session; a skill's body is free until it fires.
3. **Never restate the skill's steps in the trigger.** The rule says the skill must run; the skill owns what happens next.

Chapter 7: The Folder as Workspace

Work OS vs Knowledge Base

Summary: Two systems are routinely conflated under labels like “second brain” and “work OS”: the work OS is a workspace for active work, the knowledge base is a library that compounds topic knowledge. This guide builds the work OS; the library is an applied system in its own right.

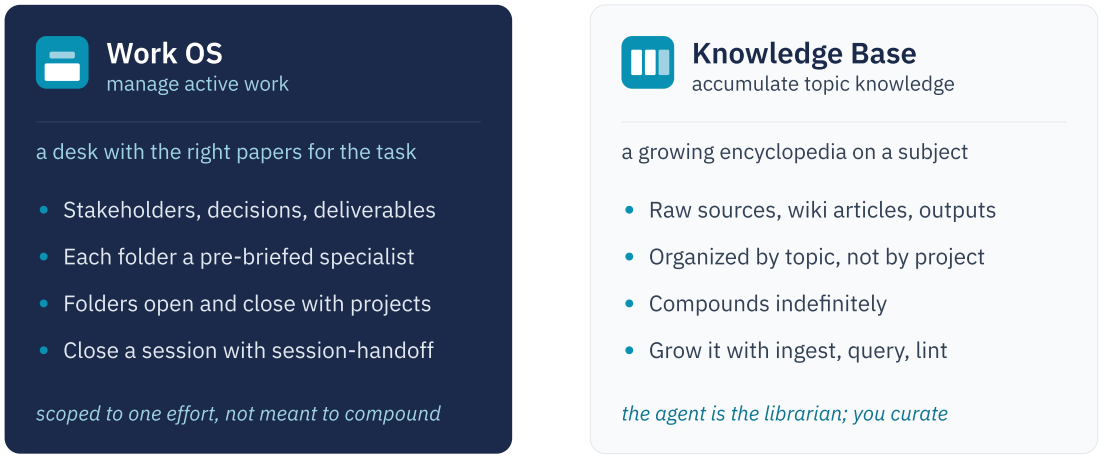
Where we are

The chapters so far gave you the problem, the agent, and the three files. Now the question is how folders compose those files into a system. Before the mechanics, there is a fork in the road that decides what kind of system you are building, and getting it wrong quietly breaks everything downstream. So this chapter opens with the distinction, then commits the rest of the guide to one side of it.

The distinction

Online content conflates two different systems under the same labels: “second brain,” “work OS,” “AI knowledge base,” “personal knowledge management.” People show folder setups and call them all the same thing. They are not. Using the wrong model produces a system that either buries operational work under knowledge-management overhead, or fails to accumulate lasting knowledge because it is organized like a project tracker.

Dimension	Work OS (workspace)	Knowledge Base (library)
Purpose	Manage active work	Accumulate topic knowledge
Content	Stakeholders, decisions, open items, deliverables	Domain insights, frameworks, patterns, synthesized understanding
Primary artifacts	Background file, working brief, deliverables	Raw sources, wiki articles, outputs
Workflow	folder-init, then work, then session-handoff	Ingest, query, lint
Lifecycle	Folders open and close with projects	Folders compound indefinitely
Sessions end with	State update (session-handoff)	Wiki update
When content is stale	Archive or close the folder	Health check, update or remove



Two systems often conflated: the work OS manages active work; the knowledge base compounds topic knowledge.

The work OS, and personal knowledge management as its discipline

A work OS is an operating system for knowledge work. Each folder is a workspace for a specific active effort: a vendor negotiation, a product build, a board document. The folder has a `CLAUDE.md`, a background file, and optionally a working brief. Work happens inside; the session closes by updating the state files. When the project ends, the folder goes dormant. Nothing in a work-OS folder is meant to compound across unrelated projects; it is scoped to one effort. The mental model: each folder is a desk with the right papers for the task at hand.

Underneath sits **personal knowledge management** (PKM), but as a discipline, not a destination. PKM here means one operating principle: organize knowledge into purposeful containers, keep what is signal, process or discard what is noise, and make sure the right context is in the right place when needed. Two practical habits carry it:

- **Capture without friction, triage with judgment.** Capture lands in the container that will own it: an artifact for a known folder goes into that folder's root, which doubles as the folder's inbox; a knowledge capture goes into the knowledge base's own intake. A staging file at the vault root exists only for what has no obvious home yet, and triage clears it against the vault map, moving each item inward or discarding it. The act of deciding where something belongs is itself a filter. An inbox is pre-curation and is never a context source for real work.
- **Curate before you execute.** Curation builds the vault ("what should live here?"); execution works from it ("what can I produce from what is here?"). Run them as separate modes. Execute against a noisy vault and the agent amplifies the noise.

That is all the PKM theory the work OS needs. The fuller "second brain" practice, a compounding library of cross-linked notes, is the *other* system.

The knowledge base, covered separately

A knowledge base follows the library pattern (Andrej Karpathy's LLM Wiki, April 2026). It is organized by topic domain, not by project. Raw source material is dropped into a `raw/` folder without organization. The agent reads it and builds a `wiki/` of cross-referenced Markdown pages with an index. Questions are answered from the wiki and saved to `outputs/`, which feed back into `raw/`, compounding over time. The agent is a librarian, not a collaborator on active decisions.

If you have met this pattern online, it was probably wearing the words "second brain": the wave of videos and guides that pair a Markdown vault (often Obsidian) with an AI agent as a compounding personal library. That wave predates Karpathy's post, but his LLM Wiki

(published as a post and a companion gist in April 2026) gave it a named, buildable shape, and much of what you now see traces back to it. It is also where this guide's author found his own starting influence for the whole work OS, which is why the pattern is taught here on its own terms rather than dismissed. So when a video shows "Claude plus Obsidian" building a second brain, you now know exactly where it sits: that is the knowledge base, the library, and not the work OS this guide builds.

This guide does not fold that pattern into the work OS. It keeps them as two systems that inform each other but do not merge, and it treats building a knowledge base as its own applied system: see Building a Knowledge Base Wiki in Chapter 11. One container can hold more than one knowledge base, one per topic domain, each with its own operating rules; Chapter 11 covers that too. This guide itself was grown inside one: a knowledge base sitting in a `knowledge-base/` container alongside the work-OS folders, not inside any of them.

Using work-OS organization inside a knowledge base (tagging every article with a project, closing folders when a topic is "done") prevents the compounding effect; knowledge never reaches the density where it generates non-obvious connections. Using library organization inside a work OS (asking the agent to synthesize domain knowledge in the middle of a vendor negotiation) produces unfocused, uncheckable output mixed with operational state. The right architecture is one work OS for active work and one knowledge-base container for topic domains, kept apart. And the router between them is intention: what the knowledge is *for* decides its destination, not what it is about. The same domain fact goes to the knowledge base when you capture it to compound, and to the work folder when it serves the active effort.

Key Principles

1. **Decide which system a folder is before you build it.** Active effort with stakeholders and deliverables is work OS. A compounding topic library is a knowledge base.
2. **PKM is the discipline, not the goal.** Capture, triage, curate. Do not build a second brain for its own sake; that is the wrong investment for this reader.
3. **Route by intention, not by content type.** The same domain fact goes to the knowledge base when captured to compound, and to the work folder when it serves the active effort. The two systems still do not merge.
4. **From here, the guide builds the work OS.** The library pattern returns in Chapter 11 as an applied system.

The Folder as the Organizing Artifact

Summary: The folder is the central artifact of the work OS: each folder is a pre-briefed specialist scoped to one area of work, folders are either dedicated or universal, they sit flat at the vault root, and the root hands its context down to every one of them automatically.

In the work OS, the **folder** is the unit of organization. You organize folders around work activities and scopes, one folder per bounded effort, and the agent behaves according to the `CLAUDE.md` in the folder it is launched from. That single fact is what makes each folder a specialist rather than a generalist:

- A `cedarline-product` folder, briefed with product scope and roadmap priorities, becomes a product advisor.
- A `vendor-negotiations` folder, briefed with counterparties and commercial sensitivities, becomes a commercial advisor.
- A `board-communications` folder, briefed with audience and approval chain, becomes a communications reviewer.

Launching in each folder gives a pre-briefed specialist, not a generalist that needs orienting each time. This is more practical and maintainable than automated multi-agent orchestration for most knowledge workers: the structure of the vault is what produces the specialists.

A naming convention keeps this navigable: folders use lowercase and dashes, no spaces or underscores (for example `vendor-negotiations`, `medence-partnership`).

Two Kinds of Folders

1. **Dedicated folders.** One bounded area of work or life: a specific project, company, counterparty, or property. Internal logic does not bleed into the rest of the vault. Most folders are of this kind.
2. **Universal folders.** Cross-cutting concerns that apply across multiple dedicated folders: a writing voice profile, email conventions, presentation standards, a master contacts file, and the shared entity intelligence covered in Shared Entity Context. They typically live at vault root prefixed with `_` to signal "shared, not a working folder" (`_context/`, `_voice/`, `_standards/`), and dedicated folders reference them when needed. A universal folder crosses folder boundaries by design; Two Kinds of Vaults, later in this chapter, says when that is safe.

The distinction decides where a new piece of context belongs. Ask: does this rule apply only inside one project, or whenever I do this kind of activity regardless of project? The first goes in the dedicated folder; the second in a universal folder, referenced from anywhere it applies. This stops you copy-pasting the same rules into every project.

A Flat Vault, and How Context Flows Down

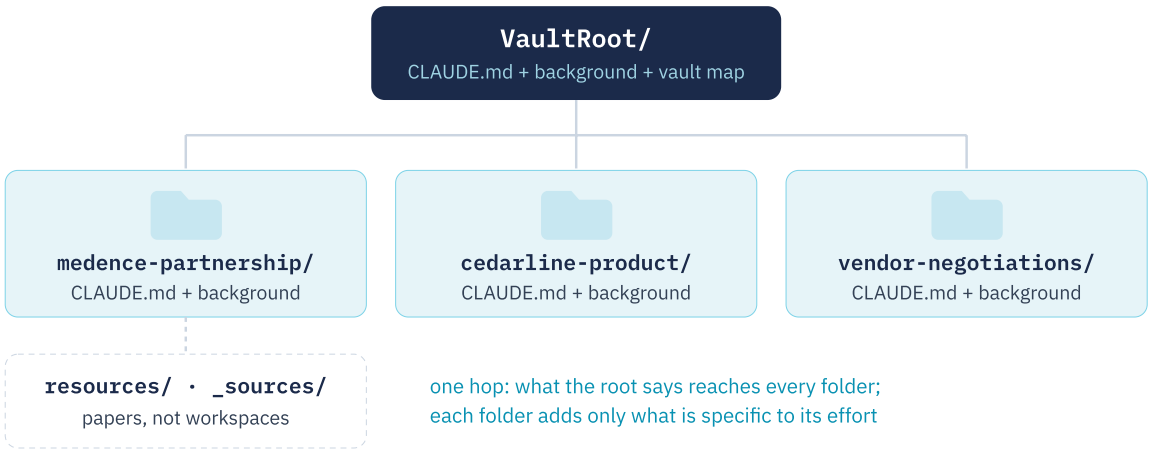
A folder never starts from nothing, because the levels above it hand their context down. Start a session inside any folder and Claude Code loads, in order:

1. `~/.claude/CLAUDE.md` - machine-global conventions
2. `VaultRoot/CLAUDE.md` - your identity and cross-project rules
3. The current folder's own `CLAUDE.md`

The agent walks up the directory tree and auto-loads every `CLAUDE.md` it finds on the way, so what you write at the root reaches every folder with no extra setup. That is the whole inheritance this system needs, and it is one hop: root to folder. A folder's own `CLAUDE.md` should be **additive and narrow**: only what is specific to this effort, never a restatement of the root. On conflict, the folder wins on folder-specific rules because it is more specific.

The mechanics would happily go deeper: folders inside folders, each layer adding rules, and the tool supports it. Resist it. Keep the vault **flat**: every workspace a folder at the root, one per bounded effort, findable in one look at the vault map. This is experience talking, not aesthetics. A workspace buried two or three levels deep stops being found; months later you will not remember which parent you filed it under, and the friction of locating it quietly kills the habit of using it. Depth also travels badly: other agent tools honor the ancestor chain when reading their standing-rules files, but not all of them reliably pick up rule files sitting below the folder they were launched in, so a deep design leans on behavior only some tools give you. A flat vault has neither problem.

What subfolders are for is papers, not workspaces: a `resources/` folder for supporting material, a `_sources/` folder for evidence, an archive for the closed and superseded. Those organize a workspace's contents and carry no rules of their own. The exceptions that genuinely earn nesting are containers whose architecture demands it, and this guide holds only two: the `knowledge-base/` container, whose sub-libraries are a deliberate structure (Chapter 11), and a staged pipeline where an item's folder is its status (an applied system in Chapter 11 works exactly this way). Both are named systems with their own rules, not filing habits.



Each folder is a pre-briefed specialist: the root's files reach every folder in one hop, and the folders sit flat, side by side, at the root.

The Vault Root Guides the Folders Beneath

In a mature vault the root is not just a container, it is the orientation layer for everything underneath. The vault-root `CLAUDE.md` instructs the agent, at every session start, to read a small set of root files:

- A vault-level background file (cross-project facts about you, key relationships, connected services).
- A terminology and invariants file (entity names, product names, codenames that override the model's training-data defaults).
- The shared entity intelligence index, if you build one (see Shared Entity Context).
- A **vault map**: one line per folder describing its scope.

The vault map matters in practice. As the vault grows past a handful of folders, you no longer want the agent listing the root directory and guessing what each folder is. The map is the maintained routing index: it is how the agent (and you) decide where a new artifact belongs and confirm whether a folder for a topic already exists, without opening folders one by one. Keeping it current is a maintenance task to fold into the session close: whenever a folder is created, renamed, or changes scope, the map entry updates in the same close.

Three root rules earn their keep as the vault grows, all learned the hard way and all written into the vault-root standing-rules file:

- **Verify against the map, never assume.** Before filing an artifact or routing a request, the agent reads the map and confirms whether a folder for the topic already exists. Folders get invented twice when nobody checks.
- **Triage first; the root is not a workspace.** Substantive work does not start at the vault root. A root session's job is routing: name the best-fit folder, move the work there, and ask once if the destination is unclear. A folder that is certainly new needs no root session at all: create it and let `folder-init` (Chapter 8) check the vault map from inside. The concierge start in Chapter 8 is this rule's create-or-extend case, for the birth that is genuinely in doubt.
- **Flag foreign content.** When something inside a folder clearly belongs to another folder's scope, the agent flags it and asks rather than silently absorbing it; it is often an accidental carry-over. One engagement, one folder, is the default this flag protects.

The root-and-folders structure is not just organizational convenience. It is a context inheritance chain, one hop deep: the root carries what is true everywhere, each folder adds only what is true for its effort, and nothing is re-specified in between. A well-structured vault composes context correctly by structure alone, so you never manually brief the agent on who you are or how you work when opening a folder. What you write once is consumed automatically, in every folder, in every session, for as long as the vault exists.

Key Principles

1. **One folder per bounded effort, flat at the root.** The folder is the workspace; scope it to one area of work and give it a place in the row, not a nest. Subfolders organize a workspace's papers; they do not create workspaces.
2. **Adding a folder requires zero edits elsewhere** beyond a one-line vault-map entry. Drop in a `CLAUDE.md` and a `-background.md` and the folder is discoverable.
3. **Write it at the root once; every folder inherits it.** A folder's `CLAUDE.md` adds what is specific to its effort; it never restates the root.
4. **Route by the vault map, not by listing the root.** The map is the durable index; keep it current.
5. **Each file written once is context you never re-explain.**

Cross-Folder Context

Summary: Two commands extend a session beyond its launch folder: `/add-dir` grants read access to a sibling folder, and `@file` pulls specific files into context on demand, with isolation as the default and access as a deliberate act.

Core Concept

Each folder is a self-contained context unit, and isolation is the design principle: your work in one folder does not contaminate another. Access is intentional: you open what a task needs and nothing else loads automatically. The work OS handles cross-folder context through three native mechanisms, no plugins or scripts:

1. **Hierarchical memory.** On start, the agent walks up the directory tree and auto-loads every `CLAUDE.md` it finds. A single `CLAUDE.md` at vault root is inherited by every folder beneath it. (This is the inheritance covered in The Folder as the Organizing Artifact.)
2. **`/add-dir` command.** Grants a running session read access to a sibling folder, without leaving the current one and without restarting.
3. **`@file` references.** Inline pointers like `@../cedarline-product/roadmap.pdf` that pull specific files into context on demand, usable inside any `CLAUDE.md` or directly in chat.

The Daily Workflow

The worked pattern, in a vault with three folders you have already met: `medence-partnership` for the engagement, `cedarline-product` for the product, `vendor-negotiations` for commercial terms. The task: from the engagement folder, draft a steering-committee memo on risks between the product roadmap and the commercial terms under negotiation.

1. **Open the session in `medence-partnership`.** The vault and folder `CLAUDE.md` files load automatically.
2. **Grant sibling access, one command per folder:** `/add-dir ../cedarline-product` then `/add-dir ../vendor-negotiations`.
3. **Load specific files and state the task in one message:** list the `@` references (including sibling `-background.md` files when current-state context is needed), then give the task.
4. **Iterate.** Follow-ups can reference more files on the fly; no need to repeat `/add-dir`.
5. **Update the memory files before closing.** This is the write side of the load/update pair: the session close, a closing prompt at first and one skill command once matured (Chapter 8). The read access you granted mid-session does not oblige a write back: whether a close ever reaches beyond its own folder is a deliberate setting, covered in Chapter 8.
6. **Close.** Nothing persists beyond the session except the files on disk.

Appendix A walks this same vault through a full folder lifecycle, end to end; this pattern is the cross-folder step of that larger run.

Command Cheat Sheet

Need	Command
Start a session in this folder	Open the folder in the desktop app, or run `claude` in a terminal there
Grant read access to a sibling folder	`/add-dir ../FolderName`
Pull a single file into context	`@../FolderName/file.ext`
Pull a folder's scope definition	`@../FolderName/CLAUDE.md`
Pull a folder's running state	`@../FolderName/FolderName-background.md`
Check accessible directories	Ask: "List your accessible working directories"
Find which folder holds a topic	Read the vault map, do not list the root
Find where new work belongs	Ask in a vault-root session; create-or-extend is its job (see Where a Folder Is Born in Chapter 8)

Supported `@` types include `.md`, `.pdf`, `.txt`, and most text formats.

Important Gotchas

1. **Sibling `CLAUDE.md` files do not auto-load.** `/add-dir ../cedarline-product` makes the folder readable but does not pull its `CLAUDE.md`. Reference it explicitly with `@../cedarline-product/CLAUDE.md` as the first item in your context list. This preserves token economy.
2. **Order matters.** Run `/add-dir` before any `@` reference to files in that folder, or the reference fails with a permission error.
3. **Path syntax.** Use relative paths (`../FolderName`). The space after `/add-dir` is required.
4. **Session scope.** `/add-dir` lasts only for the current session. Next session starts clean, which forces explicit scoping each time.

5. **Optional persistence.** Folders needed in every single session can be auto-loaded via settings; see Where to Put Configuration in Appendix B.

When This Is Not Enough

- Dozens of files from the same sibling every time: set the persistence settings.
- A cross-cutting reference used across projects: keep one canonical file in a `_shared/` folder and reference it from anywhere.
- A task that genuinely spans the whole vault: launch from `VaultRoot/` itself, where all folders are directly accessible.

Notice what repetition is telling you. In a well-scoped vault, `/add-dir` stays rare, and that is the design working, not a feature going unused. If a session in one folder keeps reaching for the same file somewhere else, the repetition is a redesign signal, and it points at one of three fixes, in order: the file actually belongs in this folder; or it carries your own conventions and belongs in a universal folder; or it is genuinely owned by the other folder, in which case declare the dependency in writing, in the persistence settings or the folder's standing rules, instead of leaving it a reflex of reaching. What repetition never justifies is moving an engagement-owned file out of its folder for convenience; in a walled vault, that is the breach again. This is Chapter 12's growth test applied to cross-folder access: friction is the trigger, and the lowest-friction durable fix wins.

Key Principles

1. **Use `@file` and `/add-dir`, never copy a file between folders to "make it available."** That is the entire purpose of this setup.
2. **Isolation is a feature.** Explicit scoping each session keeps unrelated context from leaking in.
3. **Repeated reaching is a redesign signal.** A file this folder always needs belongs here, in a universal folder, or behind a declared dependency; never leave it a habit.

Two Kinds of Vaults

Summary: Before adopting any cross-folder mechanism, classify your vault: in a one-principal vault (the executive's vault), context crossing folders compounds value; in a walled vault (the lawyer's vault), the same crossing is a breach. Every cross-folder pattern in this guide carries a label naming the kind it serves.

The mechanisms above extend a session's reach deliberately, one command at a time. The patterns from here on go further: structures and settings that move context between folders as a matter of course. Before adopting any of them, classify your vault, because the same mechanism that compounds value in one kind causes real harm in the other.

The one-principal vault. Every folder serves the same person's own work: your projects, your entities, your decisions. Call it the executive's vault: one desk, one judgment, every folder feeding it. The boundaries between folders are topics, not walls. Here, context crossing folders is pure gain. The supplier profile built during one negotiation makes the next one sharper; a decision recorded in one workstream protects its neighbor. This is the vault this guide's author runs, and it is where the mature patterns ahead (the shared entity library next, the session close that reaches beyond its folder in Chapter 8, vault-wide memory recall in Chapter 9) earn their keep.

The walled vault. Folders serve different masters: client A and client B, an employer and a personal venture, matters under separate confidentiality obligations. Call it the lawyer's vault: one practice, many masters, each matter behind its own wall. The boundaries between those folders are not organizational preference; they are ethical, contractual, or sometimes legal. Here, a mechanism that moves context across folders on its own initiative is not a feature working well. It is a breach happening automatically. At Cedarline, the internal operations folders Ken and Roni run are one-principal territory; the moment an engagement lead keeps folders for two competing clients, those two folders are walled against each other, whatever the rest of the vault looks like.

The two kinds are also not equally tested, and you should know that before relying on either. The one-principal vault is the one this guide's author runs; its patterns are lived. The walled vault's rules are built by reasoning from the obligations described above, not from practice under them. They err deliberately toward keeping mechanisms off, the safe direction to be wrong in; if you carry real confidentiality obligations, treat every walled-vault rule in this guide as a starting position and verify it against your professional duties before relying on it.

The classification has to come first because cross-folder awareness is the most impressive thing a mature vault does. It demos beautifully: ask a question in one folder and watch intelligence arrive from another. That surface appeal is exactly the danger. A walled vault that adopts these patterns for the demo effect has automated the one mistake its owner is professionally bound never to make. So each cross-folder pattern in the rest of this guide carries a label naming the kind of vault it serves, and the question to carry through the read is: which vault am I keeping? Answer it before you adopt, not after. Most vaults are mixed in practice, a one-principal core with a few walled folders, so the honest unit of classification is

the pair of folders a mechanism would connect, and one wall anywhere between them is reason enough to keep that mechanism off.

This is the second two-way classification in this chapter, and the two compose. Two Kinds of Folders sorted content by scope: dedicated to one effort, or universal across all of them. This section sorts boundaries by trust. Put together, they answer a question the walled vault will eventually ask: are universal folders still allowed? Yes, under one test. A universal folder may carry across any wall what belongs to you, your voice profile, your conventions, your standards, and may never carry what belongs to a master, meaning anything learned inside an engagement. What you learned belongs to the engagement; how you work belongs to you.

Shared Entity Context

Summary: A `_context/` layer holds reusable analytical profiles of recurring entities, and active entity matching at session start connects every folder to it through one maintained index rather than brittle per-folder reference lists.

A one-principal pattern. This is a deliberately shared, cross-folder structure: it exists so that every folder may know what any folder learned about an entity. In a walled vault, build an entity library only for entities on the same side of a wall, and never let a profile built inside one client's work inform another's.

A vault accumulates rich analytical profiles for entities you deal with repeatedly: vendors, partners, key individuals, organizations. These live in `_context/` as curated intelligence files and become a cross-session, cross-folder knowledge layer describing how those entities behave and how to work with them.

The problem: individual project folders are not automatically aware of this layer. A folder's `CLAUDE.md` lists the `_context` files it was set up with at creation. When a new context file is added later, no existing folder knows about it. The entity comes up in a session and the agent operates without the profile that already exists elsewhere. This is the invisible-context failure mode: the intelligence exists, the folder needs it, but the agent never knows to pull it.

Concretely: over a hard week of negotiation you build a profile of a supplier, how they price, where they flex, who really decides, and it lives in `_context/`. Six months later you open a new procurement folder for an unrelated project, and the same supplier turns out to be bidding. Without entity matching, the agent in that new folder knows nothing about them and you brief it from memory. With it, the profile loads the moment the supplier's name

appears in the folder's files, and the agent advises as if it had sat through the original negotiation.

The Pattern: Active Entity Matching

The fix is behavioral, not structural. Instead of relying on a static list in each folder's `CLAUDE.md`, the vault-root `CLAUDE.md` instructs the agent to perform active entity matching at session start:

1. `_context/README.md` serves as the single index of all context files, with a one-line description of each entity.
2. At session start in any folder, the agent reads `_context/README.md` and cross-references the entity names there against the folder's active documents (`CLAUDE.md`, background file, working brief).
3. Any context file whose entity appears in the folder's documents loads automatically, without the folder's `CLAUDE.md` naming it explicitly.

The result: no per-folder maintenance when a new `_context` file is added. Every folder gains awareness of the new entity the next time a session opens there. The folder's "Related Context Files" section becomes an optional pre-load hint rather than a required registry.

The Maintenance Rule

The pattern only holds if `_context/README.md` stays current. The rule is simple: whenever a `_context` file is created or updated, `README.md` is updated in the same action, not as a follow-up. If you run this pattern, make the update part of your session close: a close whose reach includes the entity library (a declared dependency, in Chapter 8's terms) enforces it automatically, and also proposes a new `_context` file when an entity has been substantively discussed, intelligence was gathered, and it is likely to recur across folders. A `_context` file with no README entry is effectively invisible to the auto-resolution pattern.

Profiles age the way folders do not, because the entity itself changes: the counterpart you profiled in one role is promoted into the deciding seat, a vendor changes hands, a partner's strategy shifts. When a session surfaces a change like that, the close that touches the entity proposes the profile update, and if the change alters what the README's one-line description says, that line moves in the same action. A profile that still describes last year's role misleads with the full authority of the entity-matching pattern behind it.

Static references rot. A folder-by-folder list of context references is a brittle copy of the index, written at setup and abandoned as the vault grows. An index maintained in one place

and queried dynamically does not rot. The `_context/` folder is your analytical memory layer, and it only functions as a layer if every session that needs it can reach it. Maintain the index; let the architecture do the rest.

Key Principles

1. **Keep explicit lists only for core entities.** For the two or three entities central to every session in a folder, an explicit entry loads them as part of session-start instructions, ahead of the matching step. Rely on auto-resolution for the long tail. Do not maintain both exhaustively.
2. **This is the horizontal axis of cross-folder awareness, and it is opt-in.** Root inheritance is the vertical axis (vault root to folder via `CLAUDE.md` traversal, tool behavior every vault gets); entity auto-resolution is the horizontal axis (vault-wide shared intelligence, a structure only the one-principal vault should build).
3. **`_context` files are stable reference intelligence, not operational state.** They are the one place where cross-linking between files earns its keep, because the files are stable and benefit from lateral navigation.

Chapter 8: Operating Day to Day

Starting a Folder: folder-init

Summary: One question comes before a new folder exists: is this certainly its own folder, or is there doubt that its scope already lives somewhere in the vault? When it is certain, create the folder and start inside it; the root is never opened. Only when create-or-extend is in doubt is the folder born from the vault root, where the concierge start decides, names, and seeds it. Either way `folder-init`, the session-start counterpart to session-handoff, orients the new folder, first as one manual prompt and later as a skill of your own: it checks the vault map for duplicate scope, challenges the name, scans existing artifacts, interviews you only about what they do not answer, and builds the context files and a source inventory before any real work begins.

A new folder for a new partner

A scene from Cedarline Advisory.

Ken came back from the conference with a backlog and a business card. The card belonged to the chief executive of a healthcare-AI company, Medence, that already ran inside a good part of the country's health sector. They had talked go-to-market for an hour and meant it. First week back, buried, an email landed: the promised product material, and a proposal to visit Cedarline's office the following week.

Now it began. An NDA to trigger with Legal and pre-sales. A pre-sales team to orient on a partnership they had never heard of. Visit dates to reconcile against a calendar Ken had not looked at properly in days. And a director colleague in Sales to bring in before any of it moved.

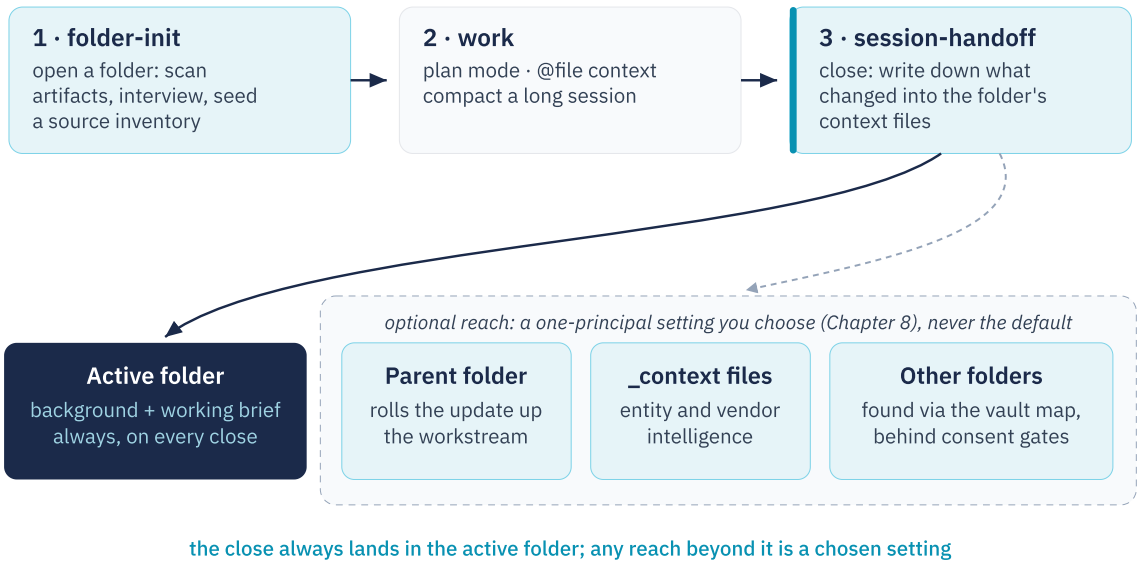
He made a folder. He named it `cedarline-medence-partnership`, the way he named all of them, and dropped in everything Medence had sent: the deck, two product videos, the email. Then he opened a session and asked the agent to read what was in the folder and tell him what it was looking at. It already knew Cedarline; that context came with the vault. It read the deck, watched the videos, and gave him back the shape of the company and where the two firms might fit. He asked it to check the proposed visit week against his calendar. It came back: that week runs into a national holiday, the following week is clear, here are two windows.

Then the package. A note to pre-sales with the material and the go-to-market case. A note to Legal clearing the standard NDA for release. A note to his director laying out the partner and why it mattered. And a digest of everything the videos and the deck contained, so pre-sales would walk in already knowing the product. Ken had reached his desk ninety minutes earlier with a business card. He left it with a partnership in motion.

The folder did the orienting. Ken did not paste a deck into a chat window and re-explain his own company; he dropped the material where the work lives and let the agent read it, against everything the vault already knew about Cedarline. How the agent reached the email and the videos is its own subject: mail can arrive as a file you drop in or through a connector your IT sets up (Chapter 10), and watching a video is a small skill an agent can be given (Chapter 12 names when a skill earns its place). That first read, turning a pile of artifacts into a briefed folder, is exactly what `folder-init` does, and it is what this section is about.

Where we are

Chapter 7 described the folder as a structure. This chapter follows a folder through time: how you open one, work inside it, and close it. The lifecycle has three moments, and a skill anchors each end. `folder-init` orients a new folder at the start. Plan mode and compact keep a working session sharp in the middle. `session-handoff` closes the session and writes what changed into the folder's files. This section covers the first. Folder-init is also the spec layer from Chapter 3 made concrete: the interview that draws your understanding out of you, before any work begins, is the spec move in practice.



The session lifecycle: open a folder with folder-init, work, then close with session-handoff, which writes the session's changes into the folder's own context files; any reach beyond the folder is an optional, chosen setting.

Core Concept

A new folder is the one place where starting from a blank page is genuinely wasteful, because the folder usually already contains evidence: documents, emails, prior versions, contracts. The evidence is there because you put it there: seeding a folder is a deliberate act, and nothing arrives on its own. `folder-init` is the discipline of orienting before building. It is the deliberate antithesis of session-handoff: handoff distributes context outward at the end; init gathers context inward at the beginning.

One thing before the mechanics, because it changes how you read them: `folder-init` is not built into the tool. It is a routine, and eventually a skill you promote yourself once the repeated open becomes tedious (Chapter 12 names that signal). What this section teaches is the practice the skill encodes, and the practice needs no installation. Its manual form is one prompt, said in a fresh session inside the seeded folder:

```
Read everything in this folder and tell me what you are looking at:
the parties involved, the apparent purpose, the current state, any
dates and decisions you can see, and what is missing. Mark clearly
what you inferred rather than found stated.
```

That is Ken's move from the scene above, and it is folder-init in miniature: orientation before work. The matured skill runs the same move in phases, with an interview and file drafts behind it:

1. **Scan before asking (Phase 0).** Before a single question, it reads the vault map and confirms no existing folder already covers this scope, checks the folder name against the vault convention (lowercase-dashes) and challenges it while renaming is still free, reads every file already in the folder, and extracts silently: named parties, apparent purpose, current state visible from artifacts, dates and decisions already made, visible sensitivities. Where a mail connector is set up (Chapter 10), it also searches the mailbox for the counterparty's correspondence, scoped to their addresses and domain; for a brand-new engagement, the mail is often the only written record that exists. It cross-references `_context/README.md` to find existing entity intelligence that matches. Then it presents a scan summary, clearly marking anything inferred rather than stated.
2. **Interview only the gaps (Phase 1).** It asks one question at a time, and only about what the artifacts did not already answer, across purpose and scope, stakeholders, domain language, current state, sensitivities, open questions, and whether the folder has enough operational complexity to warrant a working brief.

3. **Build the context files (Phase 2).** It proposes drafts and writes nothing without confirmation: a `-background.md` always (dated sections, with inferences flagged), a `-workingBrief.md` only if warranted, a `CLAUDE.md` always (load directives for the background file, the working brief if present, and any matched `_context/` files; folder scope; folder-specific rules; minimal), and the folder's one-line vault-map entry, without which no future session can discover the folder.
4. **Seed the source inventory (Phase 3).** It creates a `_sources/` subfolder with a `_source-inventory.md` table cataloguing each source by type, date, authority, and status, and a **missing context** note: what is absent that credible output would need. The gap is visible from day one.
5. **Closing report (Phase 4).** Files created, inferences to confirm, `_context/` files linked, and any open items blocking full initialization.

Manual or matured, this runs once, at the folder's orientation. Later sessions do not run it again; they open on the files it built.

Why It Matters

The quality of every later session in a folder depends on how well it was oriented at the start. A folder initialized by scanning its own evidence enters its first working session already knowing the parties, the state, and the relevant entity intelligence, instead of having you re-explain it. Flagging inferences keeps the agent honest about what it guessed versus what you confirmed, and the source inventory's missing-context note seeds the data room for the first serious deliverable before you have produced anything.

This also closes the loop opened in Chapter 6: the "let the agent draft your first `CLAUDE.md`" advice is exactly what `folder-init` operationalizes, now with an artifact scan in front of it so the draft is grounded in what is actually in the folder.

Warm Start vs Cold Start

`folder-init` and `session-handoff` are the bookends of a folder's life: init opens it, handoff closes it. The open has a subtlety worth naming, because it changes how much the agent has to work with.

A folder's first orientation has two layers, and `folder-init` captures both only if you seed before you init.

- **The durable layer** is the inventory or digest on disk: the artifacts you dropped in, and any digest the agent writes from them. It survives the session. Any `folder-init` run, today or in six months, can read it.

- **The ephemeral layer** is the live context of that first session: what you and the agent worked out in conversation while orienting. It is alive only while the session is.

Running `folder-init` after you have asked the agent to digest the seeded artifacts and the history, a **warm start**, captures both layers: the digest lands on disk and the live understanding shapes the files. Running it **cold** on an empty or unread folder captures only what you type in the interview. Hence the order: seed the folder, ask the agent to digest what is there, then init warm. The partner-folder scene that opened this section is a warm start in miniature, Ken seeded the deck, the videos, and the email first, then let the agent read before anything was built.

Where a Folder Is Born

Warm or cold, `folder-init` assumes the folder already exists. One question comes before it, and it is worth asking out loud every time a new area of work appears: **is this certainly a new folder, or is there doubt that its scope already lives somewhere in the vault?** The answer selects one of two paths, and it is the only decision that has to be made before anything is created. It does not have to be answered perfectly, either. Certainty that turns out to be wrong is caught cheaply, because the duplicate check runs at orientation on both paths, before any work has begun. Doubt is the only state the root needs to hear about.

The default path: no doubt. You know this work is its own folder. Create it, name it by the convention as best you can, drop in whatever you already have, open a session inside, and start talking. The vault root is never opened. That is what Ken did in the scene that opened this section, and `folder-init` covers the two jobs that need the whole estate: it reads the vault map and confirms no existing folder already covers this scope, and it challenges your folder name against the convention and its siblings while renaming is still a free move. Then it reads what you dropped in, searches the connected mailbox where one exists, and interviews you only about what none of that answered. Whole-estate sight was never a property of the root session. It is a property of reading the vault map, and any session can do that.

The ambiguous path: create-or-extend. You genuinely do not know whether this is a new folder or a new thread inside an existing one. That decision is about the estate, not about the folder, and there is no folder yet to decide it from, so it is root work. Open a session at the vault root and describe the work you think needs a home. Call this the **concierge start**. It does four things:

1. **It decides whether a new folder is right at all.** Often the work you are describing belongs in a folder that already exists, as a new thread inside an old scope. Create-or-

extend is the real decision at this kind of birth, and the concierge triages it against the vault map.

2. **It names the folder.** The naming convention is a root rule; the concierge applies it so you never have to recall it.
3. **It creates and seeds the folder.** The standing-rules file, the background file, the vault-map entry, all drawn from your description and whatever you drop in, following the root's file conventions.
4. **It hands you the opening prompt.** The concierge closes by writing a copy-ready first prompt for the new folder. You open a session inside the folder, paste it, and the first working session starts oriented instead of blank.

Then it stops. The root is not a workspace (Chapter 7's rule), and the cost of lingering there is concrete: a root session has no folder rules governing it, no natural place for its output to land, and no protection yet around the engagement's sensitive material. Once the destination exists, the work moves into it and the root session ends.

Before any skill exists, both paths run manually and lose nothing. On the default path, put one line in front of the orientation prompt from earlier in this section: first read the vault map and say whether an existing folder already covers this, and challenge the folder's name while renaming is still free. On the ambiguous path, the concierge start is a root session driven by the root's standing rules, delivering the same outcome the skill will later automate. When the skill arrives (Chapter 12 names the signal: a repeated open becomes tedious), the checks move into it and the two paths keep their shape.

One caution, and it matters most in the walled vault: on either path, the new folder is seeded only from what you described, what you dropped in, and what this engagement's own channels return, such as the mailbox search scoped to the counterparty. Neither the concierge nor `folder-init` ever pulls material from sibling folders into the birth, however related they look. The mailbox has its own wall risk: the same counterparty can appear in two matters, so in a walled vault a birth-time mail pull is limited to this matter's threads, not everything the counterparty has ever sent. Bringing two folders' content together is an adoption decision under Two Kinds of Vaults (Chapter 7), never a side effect of creating a folder.

Key Principles

1. **Orient first, then build.** Scanning the artifacts before interviewing means you answer fewer questions and the drafts are grounded in evidence.

2. **Mark every inference.** Anything taken from artifact scanning rather than your direct statement is flagged for you to confirm or correct.
3. **Warrant the working brief; do not default to it.** It is created only when the folder shows multiple live workstreams, active coordination, or blocking decisions.
4. **Make the gap visible from day one.** The source inventory always exists, even empty, with a missing-context note.
5. **Whole-estate sight is a property of the vault map, not of the root session.** When a folder is certainly new, create it and start inside it; `folder-init` checks the map from there, and the root is never opened. Only when create-or-extend is genuinely in doubt is the birth root work, because that decision is about the estate, and the concierge start delivers it.

Ticket, Not Prompt: The Handoff Is the Bottleneck

Summary: As your use of AI matures, the scarce resource stops being the model and becomes the handoff: the moment work leaves one worker, human or AI, and must reach the next. The fix is to hand over work orders, not prompts, and most of the work order's parts are rules this guide has already taught.

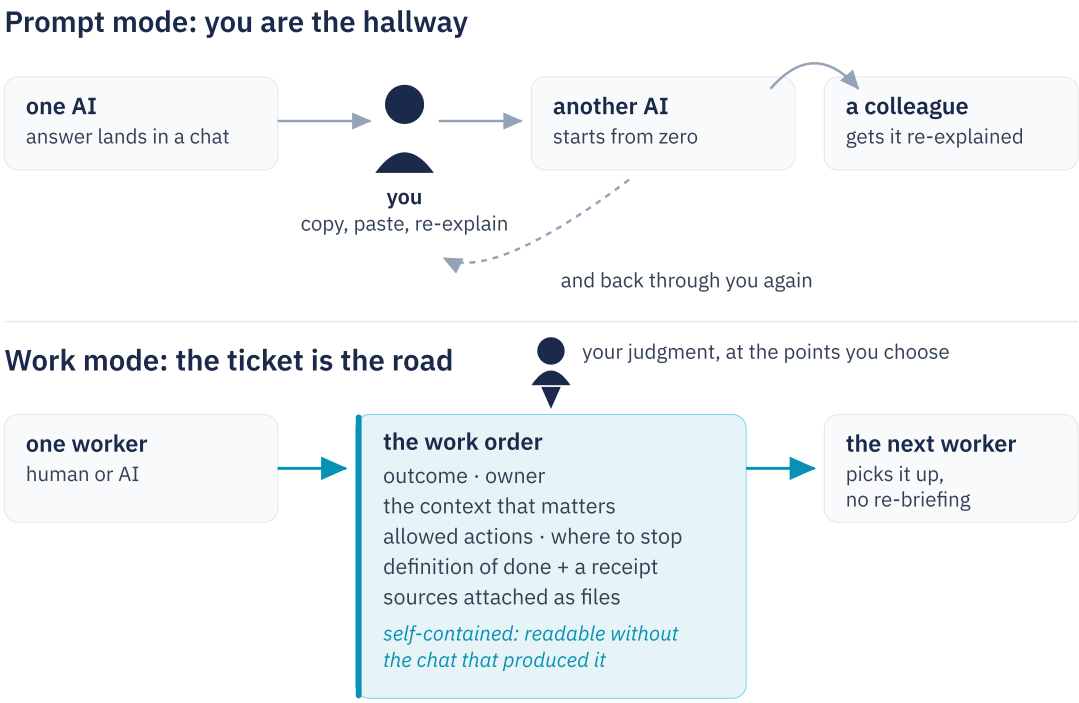
The hallway problem

Nate B Jones, whose Open Engine pattern this section draws on, names a pain most AI-productive people feel and have not named. You use several AI tools, and each does one part of the day well. But the work that leaves one tool and has to reach the next tool, or the next person, or that person's agent, has no road of its own. So you become the road: the hallway between agents, the glue, the copy-paste path. At that point the model is prompting you as much as you are prompting it.

Two of his distinctions cut sharper than the usual advice:

- **A prompt asks for an answer. A ticket asks for a result.** A prompt's output lands in a chat and dies there unless a human carries it onward. A ticket is a self-contained statement of work the next worker can pick up without reading the conversation that produced it. Because it is self-contained, workers that know nothing of each other can still coordinate through it.
- **Output is not work.** Output is what the AI returns right now. Work is something a person can review, accept, and build on. A polished brief in a private chat that nobody can find is a draft in a room by itself.

The reframe that follows: move from prompt mode (“write me a follow-up email”) to work mode (“here is the call transcript, the decision, and the constraints; draft the follow-up, flag what needs my judgment, and leave notes I can review”). The second is a full statement of work. It can be handed onward without you in the middle re-explaining it.



A prompt leaves you as the hallway carrying work between tools; a work order carries outcome, context, and stop conditions to the next worker itself.

The work order

The portable part of the pattern is a discipline, not a product: give every substantial handoff a fixed shape. Seven parts cover it: the outcome wanted, the owner, the context that matters, the actions allowed, where to stop, the definition of done plus a receipt of what was actually done, and the sources attached. (Nate’s canonical checklist runs to nine questions and sits behind his paywall; this seven-part shape is this guide’s own reconstruction from his published material. The shape is what matters, not the count.)

If you have read this far, you already own most of these parts under other names. “Where to stop” is the ask-don’t-assume rule from your standing rules. The receipt is the Sources & Gaps section your deliverables already carry. Context attached as named files is the whole discipline of not treating a chat window as state. The work order is not a new burden; it is one name and one shape for things this system already does in pieces. It is also the spec layer from Chapter 3 applied to a handoff: the understanding travels with the work, written down, instead of living in your head between tools.

And notice who the next worker usually is, because this is where the pattern stops sounding like multi-agent architecture and starts sounding like your week. For this reader the next worker is rarely another agent; it is future-you opening next week's session, or a colleague picking up the thread. You will practice the shape before this chapter ends: the closing prompt in Session Continuity is a work order addressed to your own next session, outcome, context, and stop conditions included. Appendix A step 4 shows a full one written out against a real deliverable.

The fit test: take, defer, refuse

Nate's full pattern includes infrastructure: a shared, persistent queue of tickets where agents claim work, post receipts of what they did, and stop to ask the exact blocking question when they hit ambiguity. Whether to adopt that is a fit test against your own profile, and for this guide's reader the honest answer comes in three parts.

- **Take the discipline today.** The work-order shape and the receipt cost nothing and need no new tooling.
- **Defer the queue.** A queue earns its keep under parallel, unattended work; the trigger is your first routine that runs with nobody watching. Until then it is overhead.
- **Refuse the silo.** He runs his queue on a software team's tracker. An external board breaks two rules this guide treats as load-bearing: durable state lives in plain files you own, and confidential work does not leave for a third party. When the deferring ends, the translation is the same pattern in markdown, inside the vault.

The general principle applies far beyond this pattern: ideas born in software development usually need translation for the knowledge worker. The shape transfers; the tooling often does not. This section is a worked example of making that call.

Key Principles

1. **A prompt asks for an answer; a ticket asks for a result.** Hand over statements of work, not questions.
2. **You already own the parts.** Stop conditions, receipts, and context-as-files are standing rules of this system; the work order puts them in one shape.
3. **Adopt by fit test, not by faith.** Take the discipline, defer the queue until unattended work arrives, refuse the external silo.

Plan Mode and Compact

Summary: In-session controls for high-stakes work: gate execution on an accepted plan (plan mode is that gate made mechanical), watch context utilization rather than the clock or an exchange count, take a clean exit at the ceiling, and keep `/compact` for the mid-flight case. Prevent rot first, detect it second.

Core Concept

These are mid-session controls for any working session, not just a folder's first. One governs how a task starts; the other decides how long a session stays sharp.

Plan Mode: Think Before You Execute

For tasks where the output matters (a board memo, a restructured negotiation position, a vendor comparison), the discipline that matters is a gate before execution, and it needs nothing but your prompts. Three parts: set the context first; state the goal and the deliverable; then ask for the plan, and hold delivery until you have iterated on that plan and accepted it. Only then release the agent to produce. Run this arc and you have the review gate, the verifier layer from Chapter 3 in its simplest form: you check the agent's approach against your own standard before a single file is touched, confirm the logic or redirect it, and no drafts are wasted on the wrong problem.

Plan mode is that same gate made mechanical. Enter with `/plan`, and the agent cannot write or modify files at all: it outlines its proposed approach, asks clarifying questions where intent is ambiguous, and waits for approval before proceeding. The mode adds enforcement, not intelligence. It holds the gate when your prompt forgot to ask for a plan, when your attention is low, or when the agent runs ahead mid-conversation, and it gives a newcomer the gate before the habit exists. An honest practitioner's note: run the three-part arc in every substantial prompt and you may rarely enter the mode at all; this guide's author works that way. Learn the discipline first; use the mode where the stakes are high or the habit is young. For high-stakes deliverables, save the accepted plan as a file; it becomes a record of the reasoning, useful for stakeholder review or future reference.

Compact: Managing a Long Session

A session context window has a limit. Long, iterative sessions eventually saturate it, and the agent starts losing thread, drifting from instructions, or repeating itself. `/compact` compresses the session into a distilled snapshot that retains key conclusions without the full conversational history, and the session continues from that compressed state.

Treat it as the fallback, not the plan. The primary control is the context ceiling described in the next section: watch the window's utilization, not the clock and not a count of exchanges, and take a clean exit at a natural break. Compact is for the mid-flight case, when the window is filling and the work cannot pause. Two forms:

- `/compact` compresses the full session, preserving conclusions and current state.
- `/compact Focus on [topic]` directs what to prioritize, when the session covered several topics and you want the compression centered on what comes next.

What it preserves vs loses:

Preserved	Lost
Key conclusions and agreed positions	The full back-and-forth conversation
Active task state and next steps	How each conclusion was reached
Instructions from `CLAUDE.md`	Intermediate drafts and discarded options

Why It Matters

If the reasoning trail matters (a board submission, a legal matter, a significant decision), extract the relevant reasoning to a file before compacting. Once compressed, that detail is gone. The tool will also compact on its own when a session runs near the limit, but no particular threshold is promised and the behavior has changed across versions, so do not build a workflow on being rescued automatically. `/compact` is a maintenance tool for keeping a long session functional; it does not replace the end-of-session handoff. Use compact to survive a long session; use the handoff to ensure continuity across sessions.

A related recovery tool is `/resume`, which brings back a previous session's history, useful when a session was closed accidentally or to continue same-day work that was interrupted. It is not a substitute for proper session-close context files, which cover cross-session continuity, but a quick within-day recovery. Why that recovery is always available, even after an abrupt quit, is explained in The Black Box Recorder, later in this chapter.

Context Rot: Prevent First, Detect Second

Start with the mechanic, because it explains everything else in this section. A session has no running memory of itself: on every exchange, the whole conversation so far, your prompts,

the files loaded, the tool outputs, rides back to the model inside the context window, and the model reads all of it again to produce the next reply. The window therefore only ever fills. This is not a Claude behavior; it is how these tools work, and Codex manages its sessions the same way. Two consequences follow: the longer the session, the more crowded the model's attention; and nothing you said early ever scrolls out of reach, it just competes with more and more of what came after.

Compact exists because model quality in a long session does not fall off a cliff; it erodes. First one small instruction slips. Then assumptions get bolder. Then an answer is confidently wrong. By the time the drift shows in a deliverable, the session has been rotting for a while and the wasted work is already done. One nuance is worth understanding precisely: in Claude Code your standing rules do not scroll away, and they are supplied to the model again even after a compaction. Rot is not the rules going missing. It is the model's attention thinning over a long, crowded session until rules that are still present stop being honored.

There are two ways to control a failure like that, and the difference matters more than either mechanism. A **preventive control** removes the condition that causes the failure. A **detective control** lets the failure happen and flags it early.

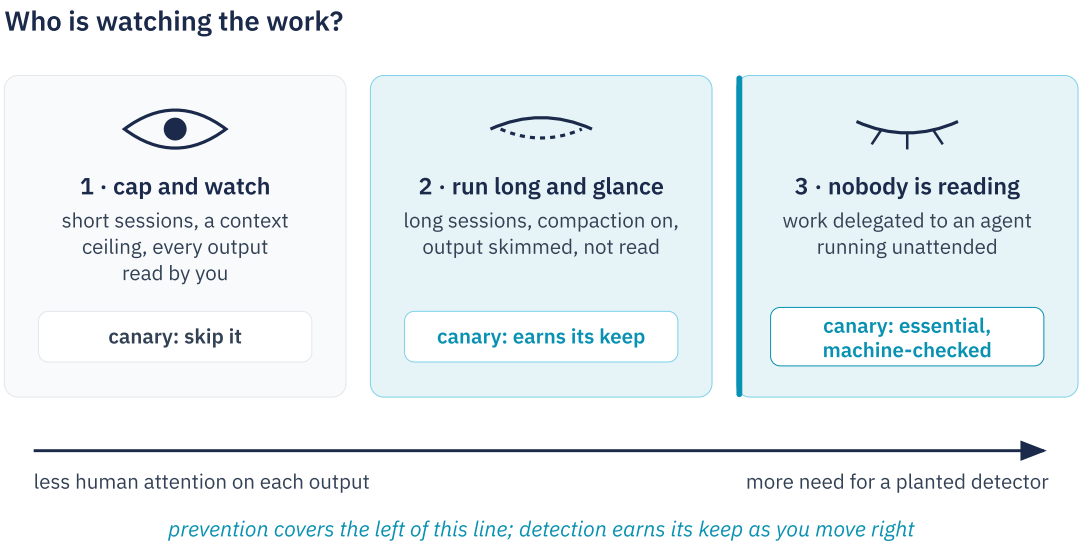
The preventive control here is a **context ceiling**: do not let a working session grow until it degrades. Treat roughly half the context window as the point to start looking for an exit, and past sixty percent, stop: run the session close and open a fresh session, which re-reads the context files and starts sharp. Where a natural break exists, this beats compacting, because a fresh session carries no summarized-conversation loss at all. Use `/compact` for the mid-flight case, when the work cannot pause; use the ceiling and a clean close everywhere else.

Keeping the number visible is what makes the ceiling workable. `/context` shows the window's current usage on demand; terminal users can put utilization permanently in view with a custom status line (`/statusline` sets one up), which is how this guide's author runs: the percentage always on screen, an exit sought as it nears half. Codex readers have the same pair, a context indicator in the interface and `/status` for the exact figure. On the desktop app surface, where less instrumentation is on screen, the on-demand command is the reliable route. Watch utilization, not the clock and not a count of exchanges; the number is the only honest signal of how crowded the session really is.

The detective control is the **canary**, a trick credited to Matt Nunogawa: plant one trivial, easy-to-check rule among your standing rules, such as "begin every reply with my name," and watch it. While the canary holds, the rules are being honored. The moment it drops, the session has begun ignoring instructions that are still in front of it, and the right move is to distrust the next answer and restart before the drift reaches anything that matters. Like the mine canary, its job is to fail first.

Which control you need depends on one question: who is watching the work?

1. **You cap and you read.** Short sessions, your eyes on every output. The ceiling plus your own reading already covers the canary's whole job, and a per-reply marker only adds noise to output you want clean. Prevention wins; skip the canary.
2. **You run long and glance.** You let sessions stretch and compaction happen, and you skim output rather than read it. Here the canary earns its keep: it catches exactly what you stopped watching for.
3. **Nobody is reading.** The work is delegated to an unattended agent. Now the canary is essential, and it cannot be a marker a human eyeballs. It must be a token that a deterministic check, a script or a hook, verifies on every output, halting the run when it goes missing. A dropped canary quarantines the work; it does not add a log line.



The canary belongs to whoever has stopped watching the work directly: redundant when you read everything, essential when nobody reads at all.

Two caveats hold everywhere. A canary proves only that the canary survived: place it beside the rules you are most afraid to lose, not at the top of the file, or it vouches for nothing that actually fails. And it detects one failure mode, instruction-dropping; it says nothing about a factual error the rules never covered.

Key Principles

1. **Gate execution on an accepted plan.** Context, then goal, then plan, then approval, then release; plan mode is the same gate enforced mechanically.
2. **Watch utilization, not exchange counts.** Keep the window's usage visible, exit near half at a natural break, and keep `/compact` for the mid-flight case.

3. **Compact loses the "how," keeps the "what."** Save the reasoning trail to a file first when it matters.
4. **Prevent rot first, detect it second.** A context ceiling and your own reading beat a canary while you are watching; the canary belongs where nobody is reading.

Session Continuity and Handoff

Summary: Session close is the write side of the load/update pair applied to a whole session: learn it first as a copy-ready closing prompt that writes the active folder's files, then mature it into the one-command `session-handoff` skill, so any later session, on any machine, opens fully oriented. How far beyond the folder a close should ever reach is a design choice, treated on its own below.

Core Concept

Sessions do not persist. When one closes, the files remain on disk but the conversational thread, what was discussed, decided, and left open, is gone. For a quick session that is fine. For extended work (long strategy reviews, multi-step deliverables, complex negotiations) losing that thread is costly, and two situations make it worse: switching machines (files sync, session history does not) and long gaps (which decisions were finalized, which are still open?). The solution is to make session state explicit and file-resident before closing.

This is the close that pairs with the open from folder-init: init gathers context inward at a folder's birth, handoff distributes it outward at each session's end.

The Manual Close, and Why to Learn It First

The simplest close is an explicit prompt: update the background file with what should persist, bring the working brief back to current if the folder carries one, and touch `CLAUDE.md` only if a standing rule changed. No separate status or handoff file: the close writes into the same files the next session already loads, which is exactly what makes it stick. But "close the session," said cold, is not that prompt; it leaves the agent guessing what a good close means. Say this instead, copy-ready:

```
Close this session. Work only inside this folder; write nothing
outside it.

1. Update the background file: add a dated section (## YYYY-MM-DD:
   Title) for what should persist from this session: decisions
   settled, facts established, stakeholder developments, artifacts
   that became important, and what is open, with the single next
   action. If a section on the topic already exists, update it and
   move its date forward instead of duplicating it.
2. If the folder has a working brief, bring it back to current:
   record the prior brief's focus in the background file first, then
   rewrite the brief to today's objective, active workstreams, open
   questions, and next moves. If there is no working brief but the
   session left this folder with several live workstreams, propose
   one; do not create it without my confirmation.
3. Touch this folder's CLAUDE.md only if a standing rule genuinely
   changed this session, and show me the change first.
4. Close with a numbered report of every file you changed and what
   the next session here will find waiting.
```

The prompt closes on the island by design: it writes into the active folder and nowhere else, which is correct in both kinds of vaults (Chapter 7), permanently so for the walled vault and the right starting setting for the one-principal one. Any wider reach is a deliberate choice, treated in How Far Should a Close Reach? below. Run this close by hand enough times to recognize what a good one looks like: what belongs in the background file, how to judge a `CLAUDE.md` change, what the working brief should say versus what is now settled. Then automate it. The day pasting the prompt becomes tedious, Chapter 12 names that signal, its text is the first draft of your own session-handoff skill: you are not adopting someone else's automation, you are promoting a prompt you already trust. One more comfort for the newcomer: Chapter 11's builder seeds this routine into the vault-root standing rules, so in a built vault even a plain "close the session" fires the full close. Learn the prompt anyway; it is the standard you review every close against.

The session-handoff Skill

The matured close is one command, and at its core it closes the active folder: the skill reads the background file, working brief, and `CLAUDE.md` together, then classifies every significant

piece of the session:

- **Background file** gets durable memory, written as dated sections (`## YYYY-MM-DD: Title`): settled decisions, stakeholder developments, milestones, established constraints, artifacts that became important. Updating existing content moves the section's date forward rather than duplicating it.
- **Working brief** (if the folder has one) is audited first: items clearly closed during the session are removed and their outcomes routed to the background file; genuinely ambiguous items (closed vs paused vs dropped) are the one place the folder close asks you; still-active items carry forward. The previous brief's focus is recorded in the background file before the brief is overwritten.
- **`CLAUDE.md`** is touched only if a standing rule genuinely changed.

That is the whole close for most vaults, and it is complete on its own: the folder's files are where the next session in that folder will look. Whatever else a close does, two steps stay constant: if you keep a vault map and a folder was created, renamed, or changed scope, the map entry is updated in the same close; and the skill ends with a numbered report of every file acted on, so you review what the close did before trusting it.

How Far Should a Close Reach?

The reach settings below are a vault-design choice, not part of the skill's definition, and anything beyond the island close is a one-principal pattern. Classify your vault first (Two Kinds of Vaults, Chapter 7).

A session sometimes produces context other folders depend on: a parent workstream moved, an entity profile gained intelligence, a fact changed that another folder's brief still states. Whether the close should carry those updates outward is Chapter 7's cross-folder question in write form, and the write form is the more consequential one. Reading across folders affects one session; writing across folders changes what every future session in the target folder believes. Three settings cover the range:

1. **The island close (the default).** The close updates the active folder only. This is the permanent setting for the walled vault, where a close that writes across folders can carry information over a confidentiality wall on its own initiative. It is also the right starting setting for everyone else, because in a vault of a handful of folders you are the propagation: you know what moved, and you can open the other folder and say so.
2. **The declared-dependency close.** The close may also update destinations named in writing in the folder's standing rules: a parent workstream's files, or the shared entity

library if you keep one. Nothing is discovered dynamically. The close follows written pointers, so no folder can quietly become a source or a target.

3. **The full-reach close.** The close may update any folder whose orientation the session changed, using the vault map to find candidates: the parent folder where a topic moved, entity profiles with their index updated in the same action, vault-level files for cross-project facts and new global rules (never the agent's auto-memory), and any other folder whose brief the session made stale. This is the setting this guide's author runs, in a one-principal vault, and even there consent gates carry the safety: the skill proposes a new entity profile and waits for confirmation, confirms before any substantial write to a folder the session did not discuss, and only flags, never rewrites, a parent `CLAUDE.md` whose general rules a child folder has outgrown.

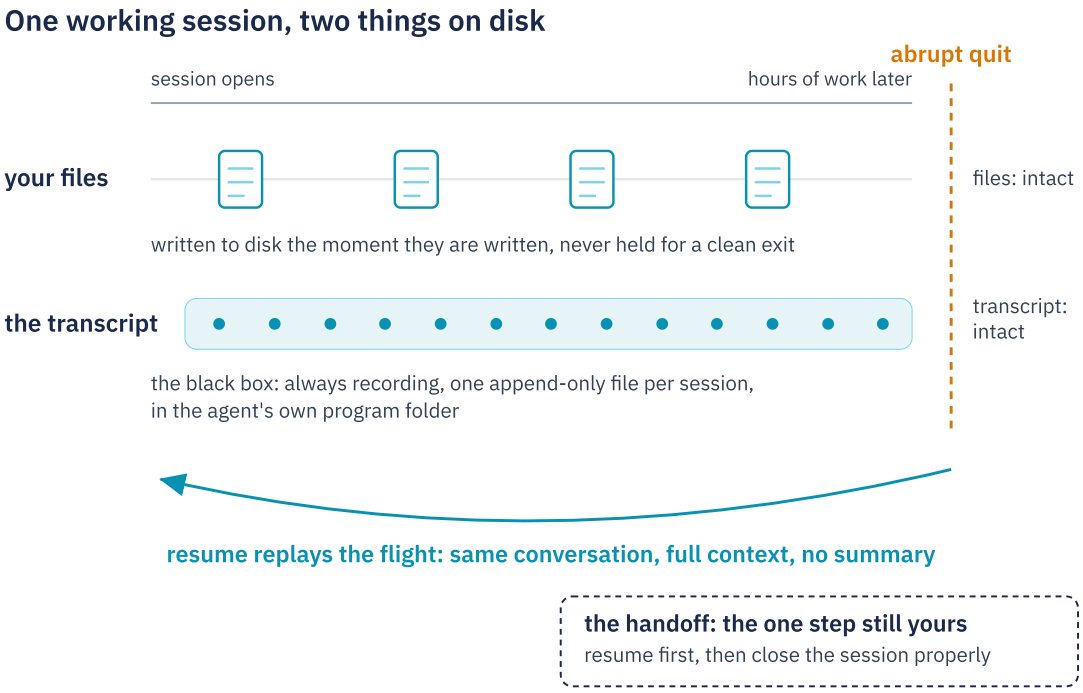
Move along the dial only when a real gap pulls you: the day you notice yourself hand-carrying the same update to the same second folder for the third time is the day to declare that dependency in writing. What does not change with the setting is the doctrine: the close always lands in the active folder, and any reach beyond it is a choice you made deliberately, recorded somewhere you can point to.

The Black Box Recorder: When a Session Ends Badly

Sooner or later you will lose an important session the wrong way: a laptop dies, a window closes, you quit by reflex with hours of dense work behind you and no handoff run. The panic says everything is gone. Almost nothing is, and knowing exactly what was at risk changes how you handle it.

Two different things are in play. **The work artifacts were never at risk.** Every document the agent created or edited was written to disk at the moment it was written, not held back for a clean exit. **The only thing genuinely outstanding is the handoff:** the step that folds the session's decisions into the background file and working brief had not run. That is the whole loss, and even it is recoverable.

It is recoverable because Claude Code keeps a black box recorder. Every session is written to disk as it runs, one append-only transcript file per session, stored in the agent's own program folder (under `~/.claude/projects/`, one directory per work folder). An abrupt quit does not delete or truncate it. The `/resume` command (introduced under Plan Mode and Compact, above) rides on this store: reopen the agent in the same work folder, resume, and pick the session, and it comes back with its full context intact, not a summary. From the terminal the same recovery is `claude --resume`, and one detail matters: the session list is scoped to the folder you launch from, so return to the folder the session belonged to first.



The transcript records the whole flight as it happens; a bad exit costs only the handoff, the one log entry a human still has to file.

The correct mental model is exactly the flight recorder: the transcript is always recording and always saved, resuming replays the flight, and the handoff is the log entry a human still has to file. So the discipline stays where this section put it, on the close, because that is the one step the machine does not do for you. But a bad exit is a two-command recovery, not a loss event. Do not start a new session in a panic; resume the old one, then close it properly. The work saves as you go; the handoff is on you.

Multi-Machine Workflow

The setup assumes the vault is synced (iCloud, Dropbox, or equivalent). The handoff is identical to a same-machine transition, with one rule: before switching machines, confirm the sync finished uploading. A partially synced vault is the one failure mode, because you work from stale files without knowing it. On the new machine, open the same folder; all `CLAUDE.md` files, background files, and working briefs carry over. As the next chapter shows, closing cleanly before switching is also what keeps the MemSearch index consistent across machines.

Key Principles

1. **The close is part of the session, not an add-on.** Skipping it means the next session starts one session stale, and staleness compounds.

2. **Durable to the background file, current to the working brief, never the reverse.** Dated sections keep the background a coherent narrative; the working brief stays a full-replace snapshot.
3. **The close always lands in the active folder; any reach beyond it is a setting you chose.** Default to the island close, and widen the reach only in a one-principal vault, along rules you wrote down.
4. **Groom standing rules and auto-memory on a slower cadence.** Update `CLAUDE.md` only on a real rule change (the six-month test); quarterly, audit auto-memory for entries that drifted or went stale.
5. **The transcript is the backstop, not the system.** A bad exit costs only the unrun handoff; resume the session, recover, then close properly.

Chapter 9: The Memory Layer

Auto-Memory

Summary: Auto-memory is a third memory mechanism, an autonomous store the agent writes on its own initiative under `~/.claude/`, useful for durable cross-folder facts about you and unhelpful for anything folder-specific.

Where we are

The chapters so far covered the memory you author: standing rules, background files, working briefs, closed by session-handoff. This chapter covers the memory layer that runs underneath and alongside that curated system, the parts that persist context without you writing every line. There are two. This section covers the agent's own autonomous store; the next covers vault-wide search.

Core Concept

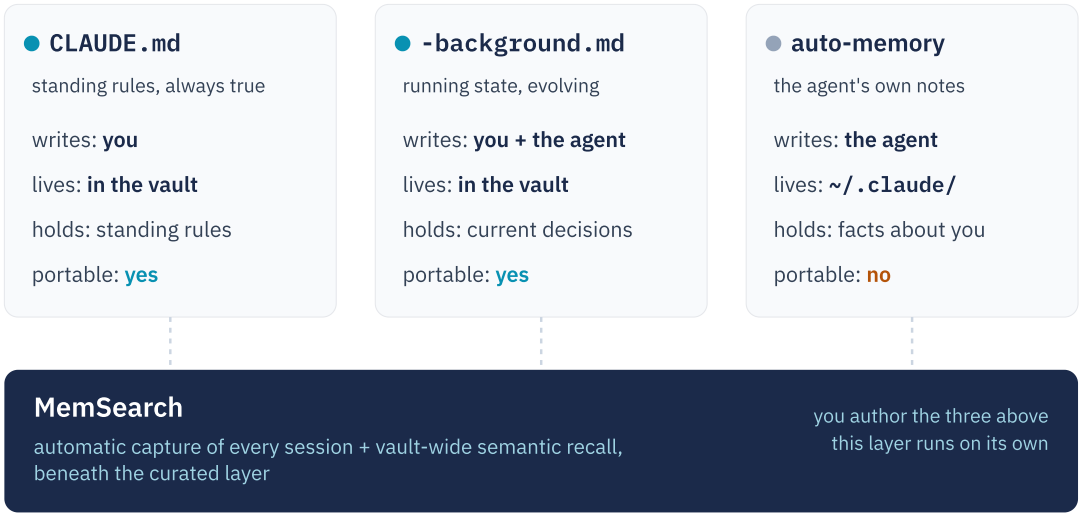
Two persistent files per folder, `CLAUDE.md` for standing rules and `-background.md` for running state, are authored by you and live in the vault. There is a third mechanism worth naming because Claude Code uses it autonomously and it can grow without you noticing. It is a separate store, kept outside the vault, with an index file named `MEMORY.md` that the agent maintains itself.

When Claude Code runs in a project folder, it maintains a private store at `~/.claude/projects/<encoded-folder-path>/memory/`, containing a `MEMORY.md` index auto-loaded into every session for that folder, and individual memory files with typed frontmatter (`user`, `feedback`, `project`, `reference`). The agent writes here on its own initiative when it judges something worth carrying forward. You audit it; you do not author it line by line.

The Three Mechanisms Side by Side

Dimension	`CLAUDE.md`	`-background.md`	Auto-memory
Lives in	Project folder, in the vault	Project folder, in the vault	Outside the vault, under `~/.claude/`
Written by	You	You, with the agent's help	The agent, autonomously
Visible in your editor	Yes	Yes	No
Loads	Every session in folder	When referenced	Every session in folder
Holds	Standing rules, always true	Running state, decisions, history	Durable behavioral context about you
Travels via vault sync	Yes	Yes	No

The defining difference is **agency**. The first two are files you author; auto-memory is the agent's notes, with you as the auditor.



Three memory mechanisms you author or audit, with MemSearch as an automatic layer beneath them.

What Belongs in Auto-Memory

Durable context about *you* that should travel across folders:

- Standing preferences (prefer Option A/B framing in all decisions)
- Behavioral feedback (avoid trailing summaries at the end of responses)
- Stable role facts (your roles across organizations)
- External reference pointers (where a particular tracker or system lives)

The test: if a piece of context applies only inside one project folder, it belongs in that folder's files. If it applies wherever you work, it belongs in auto-memory. A common mistake is putting folder-specific tone or process rules into auto-memory because the agent proactively offers to save them. They look general but are not: vendor calibration in one folder does not apply when drafting a customer pitch in another. Those belong in the folder's `CLAUDE.md`.

Note that a disciplined vault often routes *around* auto-memory deliberately: a full-reach session close (Chapter 8) sends cross-project facts to the vault-level background file and global rules to `~/.claude/CLAUDE.md`, both portable, rather than to the auto-memory store.

Why It Matters

By default the agent triggers the writes, not you. This is convenient but invisible. Six months in, an unexamined store can hold notes that are wrong, outdated, or in the wrong layer. The discipline mirrors the rest of the system: signal over noise, reviewed periodically.

You have three levers: explicit save ("remember X"), explicit refusal ("this is project-specific, not memory"), and audit-and-prune (periodically ask the agent to list what is in auto-memory for a folder, move misplaced entries, delete stale ones).

Key Principles

1. **Keep it only until MemSearch is in place.** On a single local project the native store is a fair convenience. Once you run MemSearch at the vault root it becomes a second, hidden, machine-local copy of your context, and the closing section of this chapter (Two Memory Layers, One to Keep) explains why to turn it off. Until then, audit it quarterly: ask the agent to list everything per active folder and prune.
2. **Redirect folder-specific offers.** When the agent offers to save something, ask whether it applies across folders or only inside this one. If the latter, send it to the folder's `CLAUDE.md` or `-background.md`.

3. **Auto-memory is not portable.** It lives outside the vault, so vault sync does not carry it; a second machine starts with an empty store. This is the single largest departure from the principle that everything important is a file you can see and edit, which is why the system's center of gravity stays in the vault files.
4. **Default to declining while learning.** While building the discipline of authoring your own files, decline the save offers. Auto-memory becomes a useful auxiliary later, not a replacement for the file-resident system.

MemSearch: the Memory Layer

Summary: MemSearch sits beneath the curated context system, automatically capturing every session and making it findable through vault-wide semantic recall; configured at the vault root, it searches across every folder, and it amplifies context engineering rather than replacing it.

A one-principal pattern. Vault-wide recall is cross-folder awareness on the read side: a query in one folder can surface what was said in any other. In the one-principal vault that is the entire point. In the walled vault it is the same breach the island close exists to prevent, arriving through search instead of a write: a recall run inside one client's folder can quote another client's session. If your vault is walled, keep MemSearch scoped to a single folder's history (its default behavior without the vault-root configuration below), run a separate vault per master, or skip it and rely on the folder's own files.

Core Concept

The curated system (background files, working briefs, `CLAUDE.md`, session-handoff) is a deliberate workaround for statelessness. It works well but has two structural gaps no amount of curation fully closes:

- **It only captures what you remember to save.** A rich working session preserves its value only if the close runs and the nuance was judged important enough to curate. That judgment is sometimes absent.
- **Retrieval requires knowing where the context lives.** Even when preserved, finding something two months later requires knowing which folder holds it. Cross-folder recall is impossible without manual navigation.

MemSearch addresses both without disrupting the existing architecture. Be clear about what it is first: an open-source add-on by Zilliz (released February 2026), not a built-in. Nothing about it ships inside Claude Code; installing it is an explicit act (a plugin plus a small

command-line tool), and pointing it at the vault root is a second explicit choice. Neither happens by itself. Nor is it tied to one harness: the project ships integrations for Claude Code and for OpenAI's Codex, among others, so the pattern travels even if your tool changes; the install in Appendix C is Claude Code's. Two mechanisms do the work:

- **Automatic capture.** After every agent response, a background hook summarizes the exchange and appends two to four lines to a daily Markdown file. No session-close ritual, no judgment call. The daily files are plain Markdown, readable without tooling.
- **Vault-wide semantic recall.** A skill, `/memory-recall`, runs hybrid keyword and semantic search across all daily files. The query is plain language and the index spans the whole vault, so you can be working in one folder and pull context from a session that took place in a completely different one, without knowing which.

One honest cost accompanies the convenience, and it is worth knowing before you install: a memory layer multiplies the places your work is recorded. Session summaries, the search index, and the agent's own transcripts are additional copies of whatever was discussed, and deleting a source document does not delete what the memory layer captured about it. In a one-principal vault this is a bookkeeping fact. Wherever confidentiality or deletion obligations apply, treat these stores as part of the engagement's record: know where they live (the paths are in Appendix C) and include them in whatever return-or-destroy duties you carry.

The Vault-Root Architecture

The single configuration choice that makes the "vault-wide" claim true is to store the memory at the **vault root**, not per folder. By default MemSearch writes its memory inside whichever folder the agent is opened from, which siloes each folder's history into an isolated collection that no other folder can recall. Setting one environment variable fixes this:

```
export MEMSEARCH_DIR="$HOME/.../VaultRoot/.memsearch"
```

Once set, every session, in any folder, writes to one shared collection at `VaultRoot/.memsearch/memory/`. The memory Markdown files live inside the vault and sync across machines; the vector index is machine-local and rebuilds itself on each session start from those files. This is the headline of the current setup, not a footnote: without it you have many private memories; with it you have one searchable vault memory.

One operational habit seals this: open the very first session after install from the vault root, so the hidden `.memsearch/` folder seeds at the root rather than inside a subfolder. With `MEMSEARCH_DIR` set this is belt and braces, because the variable pins the location regardless of where you launch. But understand what the variable does, because it is machine-wide: once set, every session on the machine writes to the one collection it points at, whichever folder, and whichever vault, the session was opened from. One variable serves one vault. If you keep a second, separate vault (say, one you run for another person's work), its sessions will quietly write into the first vault's memory unless the variable is unset or re-pointed for them; this guide's author found a second vault's sessions doing exactly that. In a walled setting, that is a breach arriving through configuration. The remedies, the launch pattern for a second vault, and the seeding rules are in Appendix C.

The Two Layers Compared

The curated system answers "what did I choose to preserve?" MemSearch answers "what was ever said about this?" These are different questions: the first requires judgment at close, the second requires none. They do not compete. The curated layer provides precision and structure (what you decided mattered); the MemSearch layer provides breadth and coverage (everything that happened). A reinforcing effect occurs: when recall surfaces a passage, that exchange is captured by the stop hook and re-indexed, so high-value context gains stronger representation over time without curation effort.

It Amplifies Context Engineering, It Does Not Replace It

A common misconception is that MemSearch lets you opt out of context engineering. It does not. When `/memory-recall` returns a rich answer, that depth does not come from MemSearch alone. The skill **locates** the relevant session summaries and points to where context was discussed, then **reads** the actual vault files those pointers reference and pulls the full curated content. The depth of recall is therefore a direct function of the quality of the vault files. A thin background file yields thin recall; a detailed `_context` file yields rich recall. The more disciplined the context engineering, the more valuable recall becomes. MemSearch cannot manufacture context that was never written down. The mental model: context engineering builds the memory; MemSearch makes it findable.

Three Use Cases and When to Invoke Manually

1. **Re-entering a folder after a gap.** Call `/memory-recall [topic]` before starting work to recover exploratory directions tried and dropped, interim positions revised, and numbers discussed but not formalized that may not have reached the background file.

2. **Cross-folder fact mid-session.** Something resolved in a different context surfaces. Do not guess the folder; run `/memory-recall [specific topic]` from wherever you are. The search is vault-wide.
3. **Before a commitment-level communication or meeting.** Verify the last stated position before replying to a stakeholder email or entering a negotiation. Positions drift across sessions; MemSearch does not.

It auto-triggers on obvious phrasing ("what did we decide", "last time we discussed"). For everything else the judgment is yours. Specific queries outperform generic ones: include the entity name, the topic, and a timeframe.

Key Principles

1. **The one new reflex.** When uncertain about past context, reach for `/memory-recall` before hunting through files. Outside recall moments, MemSearch is invisible.
2. **Evaluate over six to eight weeks, not one.** The index is thin early; value compounds with accumulated history. Install once the curated workflow is already habitual.
3. **The clean session-close doubles as the multi-machine optimum.** Closing one machine before opening another lets the next session index the synced memory files and recall fully from the first query.
4. **The Markdown files are the source of truth.** The index is always rebuildable from them; SessionStart re-indexes on open and the Stop hook re-indexes on close, so maintenance is mostly automatic.

Two Memory Layers, One to Keep

Summary: Once MemSearch runs at the vault root, you have two memory systems with the same name doing different jobs: the native auto-memory store and MemSearch. For a multi-folder vault synced across machines, only one earns its place. Turn the native store off, on every machine, and let the vault hold the memory.

Core Concept

Both systems are called "memory" and both write Markdown, which invites the assumption that they compete. They do not. Native auto-memory is a notebook the agent keeps about one folder and loads at startup. MemSearch is a search engine across the whole vault that you query on demand. One loads a fixed file; the other searches the entire history. Once MemSearch is in place, the question is whether to keep running both, and the answer is no.

Why the native store stops earning its place

Native auto-memory is a fair feature for a single project on one machine. A multi-folder vault synced across two machines is a different setting, and four of its properties become disqualifying there.

- **It is scoped to one folder.** Memory is keyed to a folder's path, so a session in one folder cannot see what was recorded in another. Cross-folder recall, the most useful thing in a vault of dozens of folders, is impossible by design.
- **It does not actually search.** It loads the top of an index file at startup and nothing more. There is no semantic query and no ranking. If the answer is past the load limit, you do not get it.
- **It does not sync.** The files live under `~/.claude`, not in the vault, so on a two-machine setup each machine keeps its own separate, invisible memory.
- **It breaks when folders move.** Because memory is keyed to an absolute path, renaming or moving a folder orphans everything recorded under the old path. This is not hypothetical; it is what happens the first time you reorganize.

None of this is a defect. It is a feature built for a different shape of work. Measured against the goal MemSearch was installed for, vault-wide recall that survives across machines, the native store fails the first three tests outright.

Why turn it off, not just ignore it

A second, hidden copy of your context does quiet harm. It duplicates curated context into a store you cannot see or sync, so two versions of the truth exist, one of them stale and invisible. And it runs beneath the instruction most well-kept vaults already carry, to keep durable knowledge in vault files rather than the local memory system; turning it off is what makes that instruction actually true.

The controls, in increasing order of authority: the `/memory` command (this session only), `"autoMemoryEnabled": false` in `~/.claude/settings.json` (permanent for this machine), and the environment variable `CLAUDE_CODE_DISABLE_AUTO_MEMORY=1`, which overrides both. The settings file is the natural home; the environment variable is the surest. After changing it, restart Claude Code and delete the stale memory files the feature already wrote (back them up first).

One trap, because it catches people. `~/.claude/settings.json` and the shell profile are real, separate files on each machine; they do not travel through vault sync. Disabling auto-

memory on one machine does nothing on the other. Set it on each machine independently, the same way you set the MemSearch directory and the embeddings token.

Key Principles

1. **Keep one memory layer, the one that serves the vault.** Once MemSearch is at the vault root, the native store is redundant and unsyncable. Turn it off.
2. **Turn it off on every machine.** The settings file and shell profile are machine-local; a change on one machine does not carry to the other.
3. **Let the vault hold the memory.** Durable context belongs in files you can see, edit, and sync, not in a private store under `~/.claude`.

Chapter 10: Connecting Live Context Sources

Summary: Beyond the files you author, the rest of your work context lives in your communication channels, email, calendar, chat, and meetings. Connectors let the agent read the live ones directly, and a call transcript can simply be dropped into the folder as a file. Either way, the copy-paste moment disappears.

The room had left the script

A scene from Cedarline Advisory.

The call had been going for forty minutes and it had left the script. Lama had prepared the contract clause by clause, the way you do, but the principal on the other side wanted to talk in broad strokes: the shape of the thing, where the risk sat, where each party would give. Ten million and change, several years, and now the conversation was moving faster than her notes.

Legal was on the call. Finance was on the call. Both were waiting for her to hold lines they could not all see at once.

She had spent two weeks in the contract folder with the agent. Every exchange, every position, every concession the other side had floated and pulled back, all of it was in there. So when the principal made a claim about what had been agreed in the spring, she typed one line to the agent in the folder and it came back with the actual sequence: what was proposed, when, and what replaced it. She read it back into the room as if she had it at her fingertips, because now she did. Twice more she did the same, once for a number Finance was unsure of, once for the precise wording of a fallback Legal had drafted a week earlier and forgotten. The agent was not on the call. It was in the folder, and the folder had been in every meeting that led to this one.

No chat window could have done this for her. Whatever fragments a chat product's memory might have held, it had not sat through the spring exchanges as a record she could trust under pressure; the withdrawn concession and the drafted fallback lived nowhere she could point to. Lama's agent could answer live because the whole history of the negotiation lived in the folder it was reading from. This chapter is about getting that kind of live context to the agent: connectors are one route, and a folder full of the work is another.

Where we are

Everything so far has been about context you author: the files you write and the agent reads. But a knowledge worker's day does not live only in files. It lives in email, in the calendar, in team chat, and in meetings, and that is where some of your richest, most current context accumulates. This chapter is about getting that context to the agent without retyping it. It is the last way to feed the work OS without manual effort, and it sets up the applied systems in the next chapter.

Where work context comes from

It helps to be clear about where the context for your work actually comes from. There are two origins.

The first is **you**. You are the primary source: what you tell the agent in the session, and the context files you have authored, your standing rules, background files, and the rest of the work OS. Everything in the guide so far has been about capturing and organizing that.

The second is **the work's communication channels**, the places where the rest of your working context lives and accumulates:

- **Email** and **calendar**, where most decisions, commitments, and timing sit.
- **Team chat** (Slack or a similar tool), where questions get settled day to day.
- **Meetings**, where commitments are made on a call (Teams, Zoom, and the like).

For most knowledge workers this is where the day actually happens. A decision was made in an email thread, a deadline sits in the calendar, a question was answered in a chat channel, a commitment was given on a call. None of it is in your files until something puts it there.

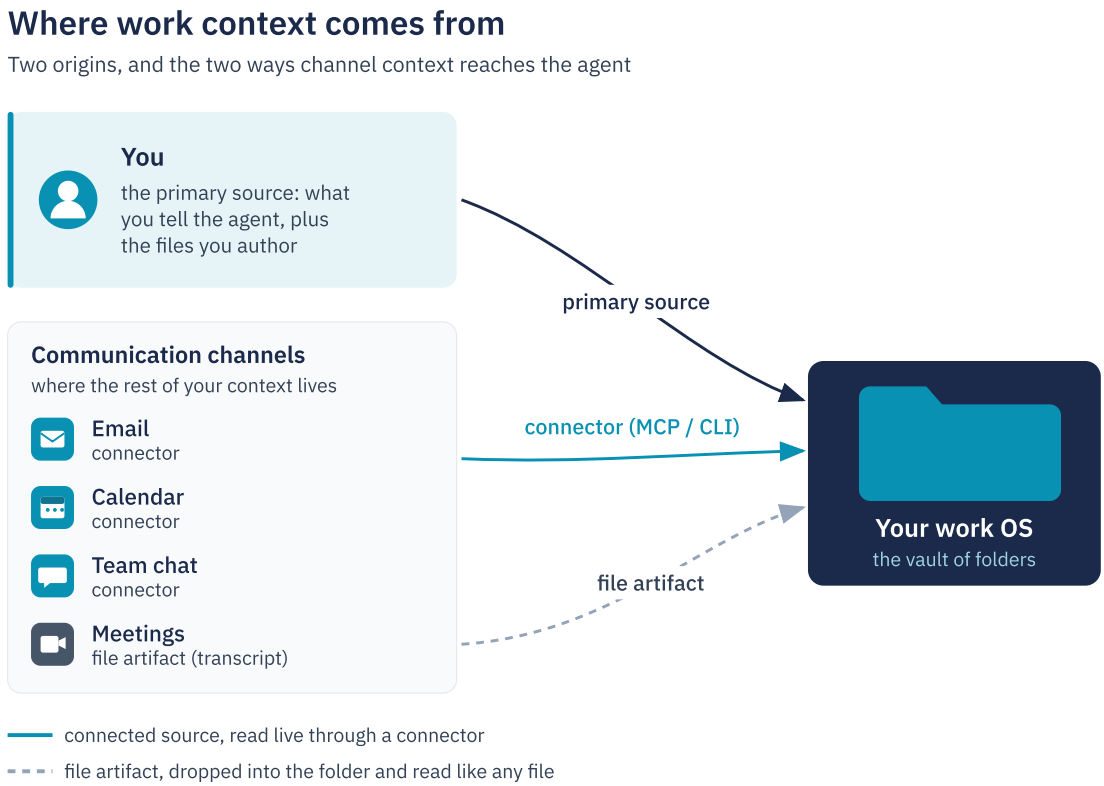
Two ways channel context reaches the agent

The job of this chapter is to get that channel context to the agent without you copying it in. It arrives two ways.

Connected sources. Live systems the agent plugs into through a connector: email, calendar, and team chat. Email and calendar use a work connector (for example a Microsoft 365 or Google Workspace connector); chat tools like Slack have their own connector too. A connector is a standard plug (often called an MCP, for Model Context Protocol) that lets the agent read a live source the same way it reads a file. You do not need to understand how the

plug works; what matters is the result: once connected, the agent reads the source directly when a task calls for it, and the "let me paste this thread in first" step is gone.

Artifact sources. Context that has already been captured as a file. The clearest case is a meeting: a transcribed Teams or Zoom call is just a text file. Drop it into the folder's workspace, name it clearly, and the agent reads it like any other file, referenced with an `@` mention or picked up from the folder, with no connector at all. This is the easiest path of all, because the transcript is already an artifact sitting where the work is. It is also a useful reminder that "live" context does not always need a live connection; sometimes it just needs to land in the right folder.



Work context has two origins, you and your communication channels, which reach the agent through a connector or as a file artifact.

Getting connected, and what to expect

You do not need to research connectors yourself; ask the agent. In a session, say what you want plainly: "Is there a connector for my work email and calendar? Find it, install it, and walk me through signing in." The agent can do the research, run the install, and talk you through the steps only you can perform, such as the sign-in screen. Setup is a one-time act per machine; once connected, you never think about the plumbing again. From then on the agent infers the route from the prompt: tell it "Legal sent an email this morning about the

settlement deed" inside the right folder, and it reaches through the mail connector, finds the thread, and folds what it learned into the session, where the close then routes the durable parts into the folder's files.

Two truths from practice, so the first friction does not read as failure. In a workplace, connecting to company email often fires an approval request to your organization's IT admin; the request is automatic, the approval is human, and the setup stalls until that person acts. Nothing is wrong with your install while you wait. And connectors read messages better than they read attachments: when a document matters, save it into the folder yourself and tell the agent it is there, and when a single email matters on its own, export it into the folder as a file, where it becomes part of the folder's record rather than something to re-fetch. Install steps for the common connectors are in Appendix B, beside the browser-tool install.

Why It Matters

Email, calendar, chat, and meetings are core context for almost any knowledge worker. A reply you need to draft depends on the thread, the relationship, and what was agreed last week. A decision about a meeting depends on what else is in the week. A follow-up depends on what was actually said on the call. None of that is in your files, and re-describing it every time is exactly the kind of manual briefing this whole guide exists to remove. Connecting the live channels, and dropping in the artifacts, lets the agent work from your real working surface instead of a summary you retype.

Channel context does not replace the file-resident system; it feeds it. A connector lets the agent read a thread or a chat; a transcript lets it read a call; the durable conclusion from any of them still belongs in a background file. The pattern is the one from earlier chapters: bring the context in, work with it, and let the close route what is worth keeping into the files. These sources widen the mouth of the funnel; the curation discipline behind it is unchanged.

In Practice

1. **Connect the live channels you want, once each.** Add a connector for email and calendar, and for team chat if your work runs on it. Setup is a one-time step, and most connectors walk you through signing in to the account.
2. **Drop call transcripts into the folder.** When you have a transcribed Teams or Zoom call, save it into the relevant folder with a clear name. No connector needed; the agent reads it as a file.

3. **Scope connectors to the folders that need them.** A connector does not have to be live everywhere. A personal-email or a particular chat workspace belongs only where it is relevant; keep each active there and disconnect it elsewhere, so the agent is never reaching into a source unrelated to the work in front of it. The routing discipline below shows how this is set once, per folder, and then holds.
4. **Triage what comes in.** Connected and artifact context both arrive raw, so treat them as input, not as filed knowledge. The agent can summarize an inbox, a channel, or a transcript; you decide what matters, and the session close routes the durable parts into the right files.

One more practice for folders that live long. A folder serving a multi-year effort accumulates artifacts by the dozen, and findability decays. A periodic housekeeping request fixes it: ask the agent to organize the folder's files into subfolders, rename them so a human can find things at a glance, and keep a small index of what lives where. The index pays twice: you find artifacts faster, and the agent pulls the right file into a session even when you did not name it. Subfolders used this way organize a workspace's papers; they do not create new workspaces, and they leave the flat-vault rule (Chapter 7) untouched. The source inventory that folder-init seeds (Chapter 8) is the same idea present from birth; housekeeping keeps it true as the folder grows.

The connector-routing discipline

You may have more than one source of the same kind, a work email and a personal email, or chat workspaces for different organizations. Give each its own connector, never cross them, and keep each scoped to where it belongs. Mixing them is how the agent ends up reading the wrong source for a task, or surfacing personal content in work output. One source, one connector, scoped to the right folders.

The scoping is set once, at folder creation, in the folder's standing rules, and from then on the same words mean the right thing in each place. In a personal-matter folder (a residency application, say), "check my email" means the personal account; in a work folder, the same phrase means the work account. Neither session can confuse the two, because each folder's rules decided before the question ever came up.

Key Principles

1. **Map your context origins.** Context comes from you and from your communication channels. Know which channels carry your real working record, usually email, calendar, chat, and meetings, and bring those in first.

2. **A connector is a means to an end.** You care about the outcome, the agent reads the source, not the mechanism. Judge a connector by whether it removes a copy-paste step you do by hand today.
3. **The easiest source is often a file already in the folder.** A call transcript needs no connector; drop it in and reference it. Not all live context requires a live connection.
4. **Channel context feeds the files; it does not replace them.** Bring it in, then let the close curate the durable parts into the work OS.
5. **Scope connectors deliberately.** Active only where relevant, disconnected elsewhere, never crossed between accounts or workspaces.

When the source is a web page

Some of your context does not sit in email or a file. It sits behind a login, on a web page: a supplier portal, a bank's reporting screen, an internal web app, a government form. The agent can reach those too, by driving a browser the way you would, reading the page, clicking, filling fields, on your behalf.

For this reader the tool to use is **Agent Browser**. It installs as a Claude Code skill, so there is no separate framework to stand up, and it is built to be light and fast. The feature that matters for business use is its credential vault: it can hold a login encrypted and let the agent use it by name, so the agent operates the page without ever seeing the password. For letting an agent into a company portal, that is the safety point that counts.

Two other tools come up in the same conversation online. The field is larger than this, but these three are its most established names, and the other two are built for software work, not for this:

Tool	Built for	When you would reach for it
Agent Browser (Vercel)	An agent driving a browser for everyday tasks	The default: open a portal, pull a report, fill a form, check a web app
Browser Use (Browser Use, the company behind the open-source project)	An agent working out an unfamiliar web task on its own	Heavier and more autonomous; rarely needed for operator work
Playwright (Microsoft)	Repeatable testing of a web app you are building	Software development, not knowledge work

The honest framing: most online coverage treats all three as developer tools, for test pipelines and scraping. Your case is narrower and cleaner, let the agent operate a web page you would otherwise click through yourself, and Agent Browser is the best fit for exactly that. The install steps and a worked example are in Appendix B. Treat the other two as names to recognize, not tools to set up.

A note on connectors versus command-line tools

A growing trend is to give the agent access to a source through a small command-line tool (a CLI) instead of an MCP connector. Most of these channels offer both: email, calendar, and chat tools like Slack each have a connector and a CLI option. For a non-technical reader this changes nothing about the outcome: the agent still reads the source. The practical reason some prefer a CLI is token economy. An MCP connector loads its full set of tool definitions into the context window at the start of every session, which spends tokens before any work begins; a CLI is called only when it is actually needed, so it costs nothing until used. If a setup guide ever offers a CLI option for a source you connect, it is a leaner route to the same result. This is a footnote, not a decision you need to make now: start with whichever connector is easiest, and revisit only if context-window cost ever becomes a real constraint.

Chapter 11: Applied Systems

Building a Knowledge Base Wiki

Summary: The companion to the work OS is the knowledge base: a topic library, built on the Karpathy LLM Wiki pattern, where you drop raw sources into `raw/`, the agent maintains a cross-referenced `wiki/`, and answers saved to `outputs/` feed back to compound the knowledge over time.

Where we are

Chapters 1 to 10 built and connected the work OS. This chapter turns to systems built on top of it. The first is the one Chapter 7 deliberately set aside: the knowledge base, the library that compounds topic knowledge rather than managing active work. It is a different system with a different workflow, and this guide itself was grown inside one.

Core Concept

A knowledge base is organized by topic domain, not by project, and the agent plays librarian rather than collaborator. The pattern (Andrej Karpathy's LLM Wiki, April 2026) has a fixed shape:

```
knowledge-base/<topic>/
  CLAUDE.md        ← how to ingest, query, and maintain this KB
  raw/             ← source material, immutable, never hand-organized
  wiki/            ← agent-maintained articles, with index.md
  outputs/         ← query answers, fed back to compound
  change-log.md    ← record of ingests, queries, health checks
```

The discipline that keeps the two systems apart, covered in Work OS vs Knowledge Base in Chapter 7, holds here: the knowledge base lives in its own container alongside the work-OS folders, never inside one, and you never edit `wiki/` by hand. The agent is the librarian; you source and curate.

Three Operations

Every knowledge-base session is one of three operations, and each is one prompt. You are the guide's reader: expert at prompting your own context by now, but new to operating a library, so the words are printed.

1. **Ingest.** Drop source into `raw/` and prompt the agent to read it and update the wiki. A single ingest touches many wiki files; let it run.

```
I dropped new material into raw/. Read it and update the wiki:
create or update the topic pages it touches, refresh index.md, add
cross-references, and log the ingest to change-log.md. Tell me
which pages changed and why.
```

2. **Query.** Ask a question; the agent answers from the wiki, with citations. If an answer contains durable insight, add it back into `raw/` so the next ingest folds it in. This feedback loop is what makes the knowledge compound.

```
Answer from this knowledge base: [question]. Start from index.md,
read the relevant wiki pages, and cite which pages and raw sources
you drew on. Save the answer as a dated report in outputs/.
```

3. **Lint.** A periodic (monthly) health check. The report goes to `outputs/`; you confirm which findings to action, then a second pass applies them.

```
Run a health check on the wiki: contradictions between articles,
orphan pages with no cross-references, claims with no source in
raw/, coverage gaps, and articles older than 90 days that look
stale. Write the report to outputs/ and stop; change nothing until
I confirm which findings to action.
```

Article Shape

Each wiki article follows one structure so the library stays consistent and AI-maintainable: a one-line summary at the top (used in `index.md`), then core concept, why it matters, key patterns or principles, cross-references to related pages, and the `raw/` sources it draws

from. `index.md` keeps one line per article. An earlier edition of this guide was maintained exactly this way, as a wiki of modular articles. When the project turned from accumulating knowledge to producing one book, the wiki stopped earning its keep and was retired for a single manuscript, which is Chapter 12's growth principle applied in reverse: a layer that stops paying for itself gets removed.

More Than One Library, and More Than One Shape

The knowledge base lives in one `knowledge-base/` container beside the work-OS folders, and the container can hold more than one: one sub-library per topic domain, say one for your field and one for management practice, each with its own operating rules in its own `CLAUDE.md`. Whether two domains share one wiki or keep two is your call: separate compounds cleaner; combined finds cross-domain connections.

And the wiki is the canonical shape, not the only one. A knowledge base that exists to produce one work can weave its material into a single manuscript instead, which is how this guide itself grew after its wiki was retired. One that exists to change how you work can run as an intake-and-triage pipeline whose outputs are working improvements rather than documents. The shape follows the library's job; declare it in that library's `CLAUDE.md`, and the agent maintains whichever form you chose.

To start one, apply Chapter 8's folder birth to a library. The first library is the one birth that reshapes the estate itself, because it creates the `knowledge-base/` container at the vault root, so a vault-root session does the honors; a later library inside an existing container follows the default path, created and initialized from inside. For the first:

```
Create a knowledge base for [topic]. Make a knowledge-base/
container at the vault root if none exists, and create [topic]/
inside it with this shape: CLAUDE.md, raw/, wiki/, outputs/, and
change-log.md. Write the CLAUDE.md with the librarian rules: raw/
is immutable and never reorganized by hand; only you maintain
wiki/ and its index.md; query answers are saved as dated reports
in outputs/; lint monthly. Add the new knowledge base to the vault
map. I will drop the first sources into raw/ when you are done.
```

Why It Matters

The work OS keeps active work current; the knowledge base lets understanding accumulate. Used together they cover both halves of knowledge work: the desk with today's

papers, and the growing encyclopedia behind it. The compounding is the point. Outputs feed back into raw, raw improves the wiki, and the wiki answers the next question better than the last. A library organized like a project tracker (closing topics when “done”, tagging by project) never reaches the density where it generates non-obvious connections.

Key Principles

1. **The agent is the librarian; never edit `wiki/` by hand.** You source and curate; the agent writes and maintains.
2. **`raw/` is immutable.** Drop sources in without organizing; never reorganize or delete by hand.
3. **Close the loop.** Outputs with durable insight go back into `raw/` so the next ingest compounds them.
4. **Lint on a cadence.** Monthly health checks keep contradictions, orphans, and stale claims from accumulating.
5. **Keep it separate from the work OS.** Topic knowledge routes here; project state stays in the work-OS folder.

A Publishing Pipeline

Summary: A four-queue pipeline where an article’s state is its folder location, its metadata travels inside the file, side work arrives as work orders, and two firewalls, career and confidentiality, gate what may publish at all; the board that reports it is a generated view, never hand-maintained.

Core Concept

Many knowledge workers publish, or want to: articles on a personal site, LinkedIn pieces, internal thought leadership. The work has a shape that chat sessions lose track of. Ideas arrive faster than drafts, drafts wait on review, finished pieces wait on dates, and every piece drags side work behind it: images, post copy, a staging step. The pipeline pattern makes all of that state physical, so no tracker document has to be believed.

The Architecture

A dedicated publishing folder carries the pipeline as four queue folders, and one rule does most of the work: **state is folder location**. An article advances by moving its file to the next queue; nothing else defines its status.

1. `1-candidates/` - ideas and drafts in development, told apart by a flag in the file.
2. `2-publishing-ready/` - finalized, both firewalls cleared, the pre-publish checklist passed. Awaiting a date.
3. `3-scheduled/` - dated and staged for publication.
4. `4-published/` - live, with the URL recorded.

Three disciplines ride on the queues:

- **Metadata travels with the file.** Each article carries a small frontmatter block of its own facts: where it came from, its stage flag, any gate blocking it, its target date. The file is the record; there is no side spreadsheet to keep honest.
- **Side work arrives as work orders.** Images, the social post, a staging task: each is handed off as a self-contained ticket (the shape from Chapter 8), and the produced assets live in a sidecar folder that moves with the article through every queue. A work order without its outputs beside it is open; with them, done. Status is physical here too.
- **The board is a generated view.** One overview file lists every queue's contents, regenerated on request from the folders and the frontmatter. It is a cache: if the board ever disagrees with the folders, the folders win. Its hand-maintained predecessor went stale, which is what taught the rule.

The Two Firewalls

Before anything is routed toward publication, it clears two firewalls, and both are mandatory: the **career firewall** (does publishing this help or harm how you are positioned professionally?) and the **confidentiality firewall** (does it leak an engagement's, an employer's, or a counterparty's material?). Anything that cannot clear both is not routed, however good the writing. The criteria live in one file the agent reads at the moment of triage, never copied into other folders, so the bar cannot fork.

A one-principal pattern, at the intake. Ideas may arrive from other folders: a working session elsewhere produces an insight that would publish well, and its close proposes routing a candidate here. That crossing is governed. Only folders that declare the pipeline in their own standing rules may route into it, the routing runs on your explicit go, and what crosses is the idea and its firewall notes, never the engagement's material. In a walled vault, the confidentiality firewall is the wall, applied at the door.

Why It Matters

The pipeline is this guide's patterns composed into one running system: folder location as state, frontmatter as metadata that travels, work orders from Chapter 8 doing real work, gates as named facts rather than memory, and a generated view replacing a hand-kept tracker. Two disciplines inherited from an earlier pipeline of the author's are worth keeping wherever you build one. **Review before write:** when the agent triages a batch of candidates, it presents the full classification as one table and writes nothing until you confirm, because classification decisions interact. And **graduate only what is proven:** nothing advances a queue on promise; it advances when the artifact is done and beside it.

Key Design Decisions

1. **State is physical.** Folder location for the article, presence of outputs for the side work. Nothing depends on a status field someone remembered to update.
2. **The metadata rides in the file.** Frontmatter travels with every move; a tracker document cannot drift from what the file says about itself.
3. **Firewalls before quality.** Career and confidentiality are gates at the door, not review comments at the end.
4. **Cross-folder intake is declared and consented.** Sources opt in by naming the pipeline in their own rules; nothing routes automatically.
5. **The board is regenerated, never maintained.** Any hand-kept overview of a live pipeline goes stale; a generated one is always the folders' truth.

Daily Briefing Agent

Summary: A morning brief you can run today with one prompt, and the scheduled agent it grows into: business context pulled from your channels into the vault on a cadence, applying session-handoff routing in reverse to complete the in-out cycle of the work OS.

Monday, 6:40 a.m.

A scene from Cedarline Advisory, of the morning this section designs toward.

Before the office is awake, Roni opens one short brief on her phone. Not her inbox, not the calendar, not six apps. One brief. What changed overnight across email and chat. What needs a decision today. What is at risk. The two or three things in her week that moved while she slept, and, because she follows the field as closely as she follows the firm, the three new videos worth her time. It was assembled at six, from her work mail, her calendar, her

reminders, and the channels she actually reads, and it was waiting before she had decided to look.

One morning the mail connector was locked out by a new security policy. The brief did not fail. The agent took the other route it had been given, pulled the threads it needed, and the brief still landed at six, with one line at the top naming the source it had fallen back to. Roni read it on the way to the kitchen and knew, before her first coffee, the two things that would otherwise have ambushed her by ten.

This is the dream most knowledge workers describe once they understand the system: a single morning brief, drawn from email, calendar, reminders, and chat, ready before the day starts, so nothing important is missed. It is not magic, and it is not quite finished; the honest design and its open questions are below, along with a version you can run by hand today. But notice what makes it possible. The brief is assembled by an agent that already knows the folders, so it can route each item to where it belongs. A chatbot has no folders to route into, and no way to run before you wake.

Core Concept

An executive managing multiple roles starts each day with context fragmented across email, reminders, and a vault of many active folders, and no structured daily awareness of what matters most. Email context reaches the vault only when manually brought into a session, so it often does not; reminders sit in a silo; nothing surfaces “what matters today” across roles before work begins; and session-handoff does nothing between sessions.

The insight: the vault already has a session-handoff pattern where, at session close, context flows from conversation into vault files across tiers. **The daily agent applies this pattern in reverse.** Instead of pushing context out of sessions into the vault, a scheduled agent pulls context from business tools into the vault on a schedule, without a human triggering it.

Together the two directions form a complete cycle:

- **Out (session-handoff):** conversation context flows into the vault at session close.
- **In (daily agent):** business context flows from tools into the vault on a schedule.

The key architectural principle: the agent uses the same routing the close already knows (folder backgrounds, the entity library if you keep one, vault-level files) to decide where each ingested item belongs. The vault structure is already the classification system.

Phase 0: The Brief You Run Yourself

Everything in the scene except “before you wake” is available today, without building anything, to any reader who has finished Chapter 10. The manual-first rule from Chapter 8 applies to agents too: run the routine by hand until you know what a good one looks like, then automate it. The morning brief’s manual form is one prompt, run from the vault root at the start of the day, where the map and every folder are in reach:

```
Assemble my morning brief. Read my unread work email and today's
calendar through the connectors, and my reminders if you have a
route to them. Give me
one brief: what changed since yesterday, what needs a decision
from me today, what is at risk, and the two or three things in my
week that moved. For anything durable, name the vault folder it
belongs to and propose the update; write nothing until I confirm.
```

Run it with the first coffee and the brief exists; only “ready before you wake” is missing. When pasting it becomes tedious, Chapter 12’s signal, the prompt is already the specification for the scheduled version. And one constraint to know before scheduling, because it shapes the whole design: a connector you signed into interactively is not automatically available to an agent running unattended on a schedule. The scheduled build has to solve access first, which is exactly why the architecture below stages its work, and why this section is honest about being unfinished.

The Architecture (Design Stage)

Two workstreams:

1. **Headless agent (intelligence layer).** A scheduled agent runs each morning: reads recent and unread email and reminders; classifies each item by folder and context type; stages a timestamped context file per batch; then a second pass applies the routing logic to update folder backgrounds, working briefs, and `_context` files, and produces a daily briefing (new context by folder plus a prioritized to-do).
2. **Dashboard app (display layer).** A project to visualize the briefing, currently design-phase only and dependent on the headless agent being proven first.

The build is phased: Phase 1 (one email account plus reminders to vault staging and a briefing file), Phase 2 (add a second email account, blocked on connector access), Phase 3 (dashboard), Phase 4 (team context sharing).

Why It Matters

A daily briefing is valuable only if it is ready before any session starts, which a manual skill cannot guarantee because it requires triggering. A scheduled agent runs on a cadence so the briefing is ready when the first session opens. Reusing the vault's tested routing instead of building a parallel classifier keeps ingested email indistinguishable from manually filed context, and two passes (a fast deterministic read-and-classify, then a judgment-based route-to-vault) reduce the risk of a long run failing partway and leaving the vault inconsistent.

Key Open Questions (as of design stage)

- Reminders integration method (which connector or CLI).
- Scheduling cadence (early-morning daily vs on-demand; weekend behavior).
- Output format (Markdown file, push notification, or rendered dashboard).
- Staging-with-review vs direct write (direct is more automated but higher risk on poor classifications).

Key Principles

1. **Run the brief manually first.** The Phase 0 prompt delivers most of the value today and becomes the specification for the scheduled agent; automate on repetition, not on ambition.
2. **The vault structure is the classifier.** Reuse the close's routing rather than inventing a parallel scheme.
3. **Prove the intelligence layer before the display layer.** The dashboard depends on the agent working.
4. **Start with the immediately accessible source.** One email account validates the architecture; the rest follow once access is resolved.

Learning Resources

Summary: For the knowledge worker, creator selection matters more than video selection; three or four channels cover the full spectrum, and this guide itself can be a source document for an AI-generated learning track.

Video is the natural starting point for anyone curious about Claude Code, but almost all of the content targets developers and software builders. A knowledge worker who searches finds tutorials that assume coding knowledge or are aimed at someone who wants to build

and sell software, which creates the impression that the tool is not for them. The gap is not the number of videos; it is creator selection. The right three or four channels give a knowledge worker everything they need.

Creator Selection

Three creators cover the full spectrum for the knowledge worker:

- **Nate B Jones (strategy and executive framing).** A 20-year product and strategy leader who frames AI adoption as a leadership and workflow decision, not a software skill. The most valuable channel for someone who thinks in strategy.
- **Brad Bonanno (workflow demonstration for non-developers).** Clear, practical, explicitly aimed at people not building software. His learning-roadmap content is more useful than any beginner feature tour.
- **Nate Herk (Claude Code mechanics).** His catalog skews toward software, but his foundational mechanics content is the most efficient way to understand how the tool actually works, without a developer-oriented course.

A Starter Sequence

A five-video order for a non-technical senior professional, each building on the prior:

1. Introducing Claude Code (Anthropic, official) - anchor the “what and why” from the source.
2. The ChatGPT-habit limit video (Nate B Jones) - the chatbot-to-agent mindset shift.
3. How I’d Learn Claude From Scratch (Brad Bonanno) - a learning roadmap, not a feature tour.
4. The core mechanics in one sitting (Nate Herk) - efficient coverage of the fundamentals.
5. AI Strategy and Frameworks playlist (Nate B Jones) - strategic framing for sustained adoption.

A note for the viewer: most tutorials demo coding because that is what the audience expects, but the tool is the same one used for pure knowledge work. Watch for the interaction pattern (conversation inside a folder, context files, iterative sessions); that pattern applies regardless of what is on screen. This is the same point made tool-neutrally in From Chatbots to Agents.

Filtering Future Content

The field moves quickly. Four questions filter what is worth your time:

1. Does the creator work in knowledge and communication, or in software development? Prioritize creators from your own background.
2. Is the video about a way of working, or a specific feature? Feature tutorials age quickly; mental models do not.
3. Does it assume technical setup or skip to the workflow? Ten minutes of terminal setup signals the wrong audience.
4. Is the creator making a strategic argument, or demonstrating a trick? Strategy compounds; tricks have a shelf life of weeks.

Using This Guide as a Learning Track

This guide can be a source document for an AI collaborator to generate a structured learning track, quiz set, or onboarding program for colleagues. The approach: share the chapters, specify the audience profile (non-technical, executive, industry if relevant), and ask for a track with learning objectives per chapter, key concepts, practical exercises, and a checkpoint question per section. The guide is written with clear section boundaries and consistent terminology to make that straightforward.

The fastest version, and the one the How-to-Read page points here for, is a tutor for yourself. Hand this PDF to any AI collaborator, paste the prompt below, and read the guide with it alongside. The prompt is written the way this guide teaches you to hand over work: outcome, context, and stop conditions included.

```
You are my tutor for the attached guide, "Context by Design." Take me
through it chapter by chapter, in order.

Before each chapter, ask me one or two questions about my own work, so
you can connect the chapter to it. After each chapter, apply its core
idea to my actual work with one concrete example, then check my
understanding with two or three short questions. Do not move to the
next chapter until I say I am ready.

If I am short on time, offer me the taste path first: Chapters 1, 3,
and 5. If I want to set the system up as we go, follow the build path:
Chapters 6 to 9 with Appendices B and C, and walk me through each
setup step at my own pace.

Keep answers short and practical. When the guide states a rule, quote
it exactly and name the chapter it lives in, so I can find it again.
If I ask something the guide does not cover, say so plainly instead of
improvising an answer in its name.
```

Using This Guide as a Build Partner

The tutor teaches you the guide; the builder puts it to work. Bound into the back of this guide as Appendix E is the **doctrine sheet**: a distillation of every chapter's summaries and key principles, written to be read by an AI collaborator handed this PDF, and generated from the same source the book is built from, so it can never drift from the text. Hand the sheet (or this PDF itself) to an AI collaborator with the prompt below, and it will interview you, position you against the guide's reader profiles and vault types, then construct your starter system, or evolve the one you already have, defaults first. The prompt carries the guide's own rules: nothing is written without confirmation, every choice ships with its default, and nothing cross-folder enters uninvited. This book can help construct what it teaches.

Running it takes three steps and no setup beyond Chapter 5. First, open an AI collaborator that can write files, Claude Code or its desktop Code surface, in an empty folder that will become your vault; a plain chat window can design your system with this prompt, but only an agent with file access can build it. Second, give it the guide: attach this PDF, and the prompt directs the collaborator to Appendix E, the doctrine sheet, as its index, with the chapters of the same file behind it for detail. Nothing else needs downloading. Third, paste

the prompt and answer the interview. The system it builds is the same one Appendix A walks through by hand, so read that appendix afterward to see what a healthy version looks like in use.

```
You are my build partner for the attached "Context by Design" guide.
Its Appendix E is the doctrine sheet: use it as your index, and pull
detail from the chapters when you need it. Your job is to construct my
work OS from the guide, or to evolve the one I already have.

Before anything else, read the copyright page and tell me this guide's
edition. Editions can correct doctrine, and the current edition is
always at omarshraim.com/context-by-design.pdf, so flag it if mine may
be stale.

Start with an interview, one question at a time. Establish: whether my
vault is one-principal or walled (Two Kinds of Vaults, Chapter 7); how
many active areas of work I carry; whether I work on one machine or
several; what AI tooling I already use; and whether anything exists
today that we should evolve rather than replace.

Then propose, before touching anything: a vault layout, the vault-root
standing-rules file (carrying two rules from Chapter 8: a folder that
is certainly new is created and initialized from inside itself, with
the root deciding create-or-extend only when genuinely in doubt; and a
session close runs the closing prompt's steps inside the active folder
only), the first two or three folders, and a draft of each folder's
standing-rules file and background file. Follow the guide's defaults unless
my answers justify
otherwise: start minimal, island close, no cross-folder mechanisms, no
memory layer yet. Where the doctrine offers a choice, name the options,
the trade-off, and the guide's default, then let me decide.

Write nothing without my confirmation, and never overwrite anything
that already exists without showing me what would change. When you
apply a rule, name the chapter it comes from. If I ask for something
the doctrine does not cover, say so plainly and flag it as our own
extension, not the guide's advice.

When the starter system is in place, close with two things: the list
of what we deliberately did not build, with the signal from the guide
that will tell me when each piece has earned its place, and a
```

```
copy-ready opening prompt for the first folder I should start
working in.
```

The five-video sequence is itself an example of the broader principle the guide teaches: know your audience, curate for their mental model, and sequence for progression. The reason this audience is underserved is not that the use case is complex; it is that the people making the content are not them.

Key Principles

1. **Curate by creator background, not by video.** Channels stay accurate longer than individual recommendations.
2. **Prioritize mental models over features.** Workflow principles compound; feature tours age.
3. **The onboarding sequence is a worked example of audience-fit curation.** Build learning paths the same way.

Chapter 12: How the System Grows

Summary: The system is never finished; it matures through use. Complexity is added only when a real session exposes a gap, and only in the lowest-friction way that closes it. This is the one idea to keep after the mechanics fade.

The closing idea

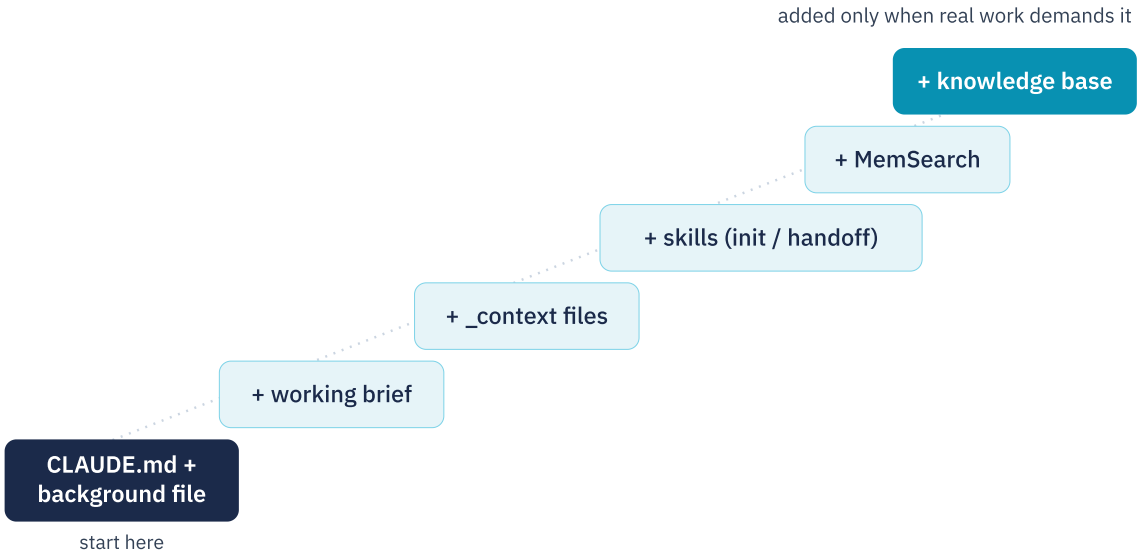
You have now seen the whole system: the problem, the agent, the files, the folder, the operating lifecycle, the memory layer, the live context sources you connect, and the systems built on top. It would be easy to read all of that as an architecture to build in one pass. It is not. The single most important thing to carry away is that the system is not designed up front. It matures through use.

Each friction point you hit in a real session becomes the trigger for a targeted fix. Complexity is added only when a real need justifies it, never in anticipation of a hypothetical future requirement. This is the design principle, not an accident of how the system happened to get built: an executive knowledge workflow that demands significant upfront architecture never gets fully built, while one that grows incrementally, session to session, from what actually caused friction, does.

This is also why the guide is sequenced the way it is. Everything after the first two files is something you add when the work asks for it:

- A working brief when a folder turns operational.
- `_context` files when an entity recurs across folders (a one-principal pattern; see Two Kinds of Vaults in Chapter 7).
- Skills like folder-init and session-handoff when a repeated open or close becomes tedious to do by hand. When one is born, the procedure moves into the skill and the trigger stays behind in `CLAUDE.md`; Chapter 6's one test decides which is which.
- MemSearch when you first catch yourself hunting across folders for something you remember discussing but never filed: the close keeps only what you judged worth saving, and this layer catches the rest (a one-principal pattern; Chapter 9 carries the walled-vault warning).
- A knowledge base when a topic keeps feeding your work, or is growing toward an output of its own. The library is the means; the payoff is what it lets you produce.

The system self-assembles from real work. This is the layer ladder named at the start of the guide, now climbed: standing rules, then running state, then cross-folder context, then live channels, then memory. Nobody built it in a day. Each rung went in when a real session asked for it, which is the only way a ladder like this gets built.



The system grows in layers, each added only when real work demands it.

This guide itself is the evidence. It describes a version of the OS that has already moved on from where it started: a manual closing prompt became a one-command handoff skill, per-folder memory became one vault-wide MemSearch collection, and the flat set of notes behind this guide became a wiki, outgrew it, and was reassembled as the single manuscript this book is built from. None of that was designed in advance. Each step closed a gap that real use exposed, and your version will diverge from this one in exactly the same way, which is the system working as intended, not drifting from a plan.

So the rule for any addition is a two-question test: did a real session expose this gap, and is this the lowest-friction way to close it? If both are yes, proceed; if not, defer. Start conservative and expand deliberately, running the manual version of a habit until you know what good looks like, then automating, the same for delegation, persistence settings, and skills alike. Friction is the trigger, never anticipation, because a feature added before a real need creates maintenance the workflow cannot sustain. If an addition starts to feel like paperwork, the work did not need it; every layer must reduce the cost of returning to a folder, and when one stops doing that, remove it. Above all, do not wait for the system to feel complete before relying on it: start with a `CLAUDE.md` and a background file, and let the rest assemble from use.

The three layers, named in hindsight

The guide has called this growth a ladder, built rung by rung in the order the work demanded. That is about the order you build in. Step back from the order and look at what you actually built, and a different shape appears, the one Karpathy named at the start.

You built a **spec** discipline: folder-init interviews you before any work begins, and plan mode makes the agent show its approach before it acts. You built a **verifier** discipline: the review gate, and the rule that no claim is accepted unless it can be traced to a source. And you built an **environment**: the standing-rules files, the folders, the memory layer, the skills, the workspace set up once and reused. The frame did not shape the guide; the guide was built from practice, and the frame fit it afterward. That is the strongest reason to trust the shape.

Be honest about the verifier, because it behaves differently for your work than for the engineering work these tools grew up in. An engineer checking the second layer has a compiler and a test suite: the code runs or it does not, and a second model can grade it well because most of what "correct" means is shared and public. Your work has neither. A memo has no compiler, and whether a position holds is a judgment whose deciding context lives nowhere but in you. Two things follow. First, your checking mostly cannot happen after the work is built, so it folds back into the spec: the review gate, where you confirm the approach before anything is written, is doing most of your verifying. Second, the one part of your work that does have a hard test is fact and citation, which is why the rule to source every claim is the single check you can enforce as strictly as an engineer enforces a passing test. Everything past that is judgment. And that is the quiet finding under the whole frame: the part of your work a machine can verify is the part you could always have handed to someone else, and the part it cannot verify, knowing what good looks like and whether the draft has reached it, was only ever yours. The method does not close that gap. It hands it back to you, which is where it belongs.

And keep the line that holds all three together. *You can outsource your thinking, but you cannot outsource your understanding.* Everything in this guide is machinery for one purpose: to move your understanding into a form the agent can act on, and to keep the one thing that does not commoditize, your judgment about what matters, firmly yours.

Where to go next

For a single concrete pass through the matured system end to end, read Appendix A: Worked Example. For setup, Appendix B covers persistent settings and Appendix C covers the MemSearch install.

Appendix A: Worked Example

Summary: One concrete pass through the matured system end to end: open a folder with folder-init, hand the work over as a work order gated by plan mode, keep the session inside its context ceiling, then close with session-handoff, against a vault whose root files and vault map orient every session and whose MemSearch layer captures it all.

Why this appendix exists

The chapters explain each part in isolation. This walks the whole system as a single sequence, using the matured skills rather than the manual prompts the earlier chapters teach first. The scenario continues Ken's partnership from Chapter 8: Cedarline is opening a folder for the Medence engagement and producing its first briefing.

Step 1: Create the folder, then run folder-init

A partnership with a company the vault has never seen is certainly its own folder, so this is the default path from Where a Folder Is Born (Chapter 8), and the vault root is never opened. Create `cedarline-medence-partnership`, named by the convention, and drop in whatever you already have: an intro email, a prior proposal, a term sheet draft. Then open a Claude Code session in the folder and run `/folder-init`. (Had there been doubt, say an existing healthcare-alliances folder that might already own this scope, the birth would have been root work instead: a concierge start deciding create-or-extend, then handing the work in.)

What happens before you answer a single question:

- It reads the vault map and confirms no existing folder already covers this engagement.
- It confirms the folder name fits the convention.
- It reads every file already in the folder and extracts the named parties, the apparent purpose, the current state, any dates or decisions, and visible sensitivities.
- Where a mail connector is set up (Chapter 10), it searches the mailbox for Medence correspondence, scoped to their addresses; for a brand-new engagement, the mail is often the only written record there is.
- It reads `_context/README.md` and flags any existing entity intelligence that matches the parties it found.

- It presents a scan summary, marking clearly what it inferred versus what was stated.

Then it interviews you only on the gaps, one question at a time: scope, stakeholders, domain language, current state, sensitivities, open questions, and whether the folder has enough live complexity to warrant a working brief.

Step 2: Confirm the context files

`folder-init` proposes drafts and writes nothing without your confirmation:

- `cedarline-medence-partnership-background.md` - durable context in dated sections, inferences flagged.
- `cedarline-medence-partnership-workingBrief.md` - only if the interview showed parallel workstreams or blocking decisions.
- `CLAUDE.md` - standing rules only, with load directives for the background file (and working brief if present) and for any matched `_context/` files, plus the folder's scope and sensitivities.
- The folder's one-line entry in the vault map, without which no future session can discover the folder.

It also seeds `_sources/_source-inventory.md` cataloguing what you dropped in, with a missing-context note naming what a credible briefing would still need. Review and confirm. The folder is now a pre-briefed specialist.

Step 3: Open the working session

Close and reopen a session in the folder (or continue). At session start the agent loads, automatically and in order: `~/.claude/CLAUDE.md`, the vault-root `CLAUDE.md` (which pulls the vault background, the terminology invariants, the `_context` index, and the vault map), the folder `CLAUDE.md`, and, by its instruction, the folder background and working brief. Any `_context` entity named in the folder's files loads through entity auto-resolution. You have typed nothing yet and the agent is fully oriented. Underneath, MemSearch is capturing the session to the vault-root memory collection.

Step 4: Hand over a work order, gated by plan mode

For a deliverable that matters, do not toss a prompt; hand over a work order (see Ticket, Not Prompt in Chapter 8), and enter plan mode so the approach is confirmed before any file is

touched:

```
/plan
Work order: a one-page engagement briefing for the Medence partnership.
Outcome: a briefing Ken can forward to his director unedited, covering
where the partnership stands, the open commercial questions, and a
recommended next move, in Situation / Key Insight / Options /
Recommendation form.
Context: @_sources/medence-term-sheet-draft.md plus the current
background file.
Allowed: read this folder and _sources; write one new file,
medence-briefing-v1.md.
Stop and ask if the term sheet contradicts a decision recorded in the
background file.
Done when the briefing covers all three points and ends with a Sources
& Gaps section as the receipt.
```

Confirm or redirect the plan, then release execution. Notice what the shape bought you: the context travels as named files rather than pasted text, the stop condition protects a decision you already made, and the receipt will show you what the briefing leaned on and what is still missing.

If the iteration runs long, watch the context, not the clock. Where a natural break exists, take the exit the guide recommends: close with the handoff below and reopen fresh, inside the context ceiling (see Context Rot in Chapter 8). Mid-flight, when the work cannot pause, compact instead:

```
/compact Focus on the agreed briefing structure and the commercial decisions
made
```

Step 5: Close with session-handoff

Run `/session-handoff` (see Session Continuity and Handoff in Chapter 8). In one command it:

- Writes settled decisions to the background file as dated sections, audits the working brief (routing closed items to the background, carrying active ones forward, recording

the prior brief's focus first).

- Touches `CLAUDE.md` only if a standing rule changed.
- Because Cedarline's operations vault is one-principal and runs the full-reach close (Chapter 8), it then propagates what is material: it updates or proposes a Medence `_context` profile since the partner was substantively discussed, sends any cross-project fact to the vault background, and updates any other folder whose orientation changed, using the vault map to find candidates. On the island close, this step simply does not run.
- Updates the folder's vault-map entry if its scope shifted since birth, and ends with a numbered report of every file it touched.

Step 6: Confirm and move on

Review the closing report. Save the briefing with a clear name. The next session, on this machine or another after sync completes, opens fully oriented, and `/memory-recall medence partnership commercial questions` will surface this session from any folder in the vault.

And if the session had ended badly instead, a quit before the close, nothing above is lost: the briefing was on disk the moment it was written, and the conversation is in the transcript. Return to the folder, `claude --resume`, then run the close you missed (see The Black Box Recorder in Chapter 8).

What this demonstrates

Every part of the guide in one run: the files, the folder and its inheritance, the vault root and map as the orientation layer, the work handed over as a self-contained order and returned with a receipt, the session kept inside its ceiling, the lifecycle skills at both ends, the memory layer underneath. You authored very little of the orientation by hand; the architecture supplied it.

Appendix B: Environment, Setup, and Browser Tools

Summary: `settings.json` offers two levers for persistent cross-folder context, auto-loading sibling `CLAUDE.md` files and auto-granting folder access, but the default workflow needs neither; add persistence only when a session pattern repeats often enough that typing `/add-dir` becomes real friction.

Core Concept

Beyond `CLAUDE.md`, Claude Code reads two kinds of configuration: environment variables (named flags checked at launch) and `settings.json` (a config file persisting the same flags plus permissions, pre-granted directories, and model choice). For the executive workflow you almost never need either. They become useful only when a specific session pattern repeats enough that the friction of typing `/add-dir` every time outweighs the benefit of session-by-session scope control. This is the incremental design principle: let friction surface before configuring around it. The specifics in this appendix were last verified in July 2026; settings names and connector flows evolve, so where this page and the tool's current documentation disagree, trust the documentation.

Where to Put Configuration

Two places. Shell environment variables are system-wide and affect every session, but editing shell profiles is technical and not recommended unless you are comfortable with it. `settings.json` is scoped, clean, and recommended, with three locations of increasing specificity:

1. `~/.claude/settings.json` (user-wide)
2. `VaultRoot/Folder/.claude/settings.json` (project-wide, shared if committed)
3. `VaultRoot/Folder/.claude/settings.local.json` (project-wide, personal, auto-excluded from version control)

For a personal vault like this one, use option 1 or option 3. The recommendation is `settings.json` over shell exports: a single file you edit once, with no shell-profile editing or OS-specific syntax.

The Two Levers

1. **Auto-load sibling `CLAUDE.md` on `/add-dir`.** By default, `/add-dir ../cedarline-product` grants file access but does not load `cedarline-product/CLAUDE.md`; you pull it with `@../cedarline-product/CLAUDE.md`. To always auto-load, set `CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD` to `1` in the `env` block. Trade-off: convenience versus token economy, since every `/add-dir` then pulls a full scope file even when the task needs one specific file.
2. **Auto-grant access to specific siblings on launch.** Pre-declare folders in `permissions.additionalDirectories` (for example `../cedarline-product`, `../vendor-negotiations`) in a folder's `settings.local.json`. Every session from that folder then has those siblings accessible immediately. You still pull individual files with `@`. Trade-off: saves typing, loses session-by-session explicit scoping. Recommended for siblings referenced in 80%+ of that folder's sessions.

Combining both is the "heavy persistence" pattern: lowest friction, highest baseline context cost.

Why It Matters

The more persistence you add, the less explicit your session scope becomes. That is a real trade-off, not just a preference. Explicit scoping is a feature: it keeps unrelated context from leaking in and keeps sensitive folders accessible only on deliberate opt-in.

Key Principles

1. **Start with no settings.** Work the default flow for a few weeks and let friction surface organically.
2. **Prefer `settings.local.json` and folder-scope over user-wide.** The `.local` variant is excluded from version control, and scoped beats global, mirroring signal over noise. Keep `~/.claude/settings.json` minimal.
3. **Document persistence inline.** JSON has no comments, so add a note at the top of the folder's `CLAUDE.md` explaining why siblings are pre-loaded.
4. **Re-evaluate quarterly.** Persistent settings drift; folders used constantly six months ago may be cold now.
5. **Avoid persistence for sensitive folders, large or binary content, and short-lived projects.** Verify active settings with `/status`, which also flags malformed JSON (the

most common failure).

Setting Up Your Environment: Warp and Speech-to-Text

Chapter 5 recommended Warp as the place to run the agent, and Spokenly (or SuperWhisper) for dictation. The granular steps:

Warp. Download Warp for macOS and install it like any app. Open it and you have a terminal with tabs, a folder explorer, and an editor-like surface. Start a Claude Code session by typing `claude` in a Warp tab pointed at your vault folder. Warp's own AI assistant is built in: when you do not know a command, describe what you want in plain English on the command line and let it propose the command. This is the route used to update the Claude Code plugin on the day a new model shipped, with no documentation hunt.

Speech-to-text. Install Spokenly from its site, or SuperWhisper as the alternative. Grant it microphone and accessibility permissions in System Settings, Privacy & Security, then set a push-to-talk or toggle shortcut. From then on it dictates into Warp and into any other macOS application, which is what makes giving the agent a full spoken prompt as cheap as typing a short one.

Connecting Email, Calendar, and Chat

Chapter 10 recommended connecting your live channels through connectors. The path of least resistance is to let the agent drive its own setup:

1. In a session, ask: "Find the connector for [Microsoft 365 / Google Workspace / Slack], install it, and walk me through sign-in." The agent locates the connector, runs the install steps it can, and hands you the ones only you can do, typically a browser sign-in to your account.
2. Expect the corporate pause: work accounts often require an IT admin to approve the connection request. The request is sent automatically; the approval is human and can take days. Nothing is broken while you wait.
3. Repeat per machine: connectors are machine-local, like the settings in this appendix, so a second machine needs the same one-time setup.
4. Scope it: enable the connector where the work needs it and keep it off elsewhere, per the routing discipline in Chapter 10.

One caveat worth knowing from day one: connectors read messages well and attachments poorly or not at all. Files that matter get saved into the folder and named to the agent.

Installing Agent Browser

Chapter 10 recommended Agent Browser for letting the agent operate a web page. It installs as a Claude Code skill, so setup is light:

1. Add the skill the way you add any Claude Code skill or plugin (`/plugin` inside a session, or the skill's documented install command), then reload.
2. On first use, the agent opens a managed browser. Sign in to a site once and store the login in Agent Browser's credential vault, so the agent can reference it by name on later runs without ever seeing the password.
3. Scope it like a connector: enable it where the work needs a web source, and leave it off elsewhere.

A worked use: ask the agent, from inside the relevant folder, to open a supplier portal, pull the latest statement, and summarize what changed since last month. It drives the browser, reads the page, and writes the summary into the folder, with no copy-paste from you. Treat Browser Use and Playwright (named in Chapter 10) as developer tools you can ignore unless you are building or testing software.

Appendix C: MemSearch Installation

Summary: A no-coding setup for MemSearch on macOS: install the CLI and plugin, configure local ONNX embeddings and Sonnet summarization, set `MEMSEARCH_DIR` to the vault root for vault-wide recall, and verify with a fresh-session capture and a cross-folder recall test.

Core Concept

MemSearch adds automatic capture and vault-wide recall beneath the curated context system (see MemSearch: the Memory Layer, Chapter 9). Vault-wide recall is a one-principal pattern: read the label at the top of that Chapter 9 section, and Two Kinds of Vaults in Chapter 7, before pointing a walled vault at one shared index. Setup takes 30 to 45 minutes and requires no coding. It runs on local, free components except the summarization hook, which costs roughly $3 to $8 per month.

Two honesty notes before the steps. The procedure is macOS through and through, Homebrew, `~/.zshrc`, and macOS privacy settings included; on Windows the same components exist, but the steps differ and this appendix does not cover them. And the commands and defaults below were last verified in July 2026 against the project's current release; it moves quickly, so where this page and the project's own documentation disagree, trust the documentation.

Prerequisites

- **Claude Code authenticated via claude.ai** (not a standalone API key). The capture hook calls `claude -p`, a Claude Code subprocess that uses the existing claude.ai OAuth session.
- **Homebrew** (the macOS package manager).
- **A HuggingFace account with a Read token.** MemSearch downloads a local embedding model that requires a free account and token. Create the token under Settings, Access Tokens, type Read.
- **About 558 to 570 MB of free disk space** for the model, cached once at `~/.cache/huggingface/hub/`.

- **macOS Terminal with Full Disk Access** (System Settings, Privacy & Security) so memsearch can read files in Documents.

Setup Steps

1. **Set the HuggingFace token** in `~/.zshrc`: `export HF_TOKEN=your-token-here`, then `source ~/.zshrc`. Treat the token like a password: it now sits in plain text in your shell profile, so never share or paste that file anywhere, and if the token ever leaks, revoke it on the HuggingFace site and issue a new one. Do not set `ANTHROPIC_API_KEY`; a shell-level API key causes an auth conflict warning every session and is not needed.
2. **Install uv:** `brew install uv`.
3. **Install the MemSearch CLI.** Requires Python 3.10+. macOS ships 3.9, so check `ls /opt/homebrew/bin/python3*` and pass the newer one: `uv tool install "memsearch[onnx]" --python /opt/homebrew/bin/python3.14` (substitute your version). The `[onnx]` flag installs the local embedding runtime and is required. Add the binary to PATH: `echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc && source ~/.zshrc`.
4. **Install the plugin inside Claude Code:** `/plugin marketplace add zilliztech/memsearch`, then `/plugin install memsearch` and choose **"Install for you (user scope)"** so it is active across all folders, then `/reload-plugins`. Successful registration shows as 10 hooks.
5. **Configure** in the terminal:

```
memsearch config set plugins.claude-code.summarize.model claude-sonnet-4-6
memsearch config set embedding.provider onnx
memsearch config set embedding.model "gpahal/bge-m3-onnx-int8"
```

Verify with `memsearch config list`.

The Two Configuration Points That Cause Most Failures

1. **`MEMSEARCH_DIR` must point at the vault root, or recall is not vault-wide.** By default memsearch stores memory inside whichever folder the agent is opened from, siloing each folder's history into an isolated collection. Add to `~/.zshrc` on every machine:

```
export MEMSEARCH_DIR="$HOME/Documents/VaultRoot/.memsearch"
```

Once set, all sessions write to `VaultRoot/.memsearch/memory/` regardless of which folder is open. The memory Markdown files live inside the vault and sync via iCloud; the vector index is machine-local and rebuilds on each session start. This is the architecture, not a tweak: it is what makes the "vault-wide" claim in the main chapter true.

2. **The embedding model ID must be the explicit `gpahal/bge-m3-onnx-int8`.** The short form `bge-m3` and the full path `BAAI/bge-m3` both fail on a fresh install. The short form appears to work only on a machine where the correct model is already cached; on a clean machine it downloads from the wrong repo and fails.

3. **A second vault needs the variable re-pointed, not just a root session.** `MEMSEARCH_DIR` is machine-wide: when it is set, every session on the machine writes to the collection it points at, regardless of which vault the session was opened from. A second vault kept separate therefore needs its sessions launched with the variable pointed at its own root. The clean pattern is a per-vault launch alias in `~/.zshrc`, for example `alias secondvault-claude='MEMSEARCH_DIR="$HOME/Documents/[second-vault]/.memsearch" claude'`, used for every session in that vault. (Running with the variable unset instead makes memsearch isolate per launch folder; that works, but then the first session in the vault must come from its root, or the memory anchors inside a subfolder.) If a second vault's sessions have already leaked into the main collection, move its entries out of the main `memory/` daily files into the second vault's own `memory/` before the next session, and re-index both collections.

Verification

Open a completely fresh session (the SessionStart hook only fires at the start of a session). Have four to five real exchanges on actual work, then confirm the daily file holds substantive two-to-four-line summaries:

```
find ~/Documents -name "$(date +%Y-%m-%d).md" -path "*/.memsearch/*"
2>/dev/null
```

Then open a session in a different folder and run `/memory-recall [topic from those exchanges]`. It should return passages from the other folder's session, confirming vault-wide recall.

Multi-Machine Notes

The daily `.md` files sync via iCloud because they live in the vault. The vector index, config, and model are machine-local and set up per machine. The right workflow for most users: close one session cleanly before opening another on a different machine. On open, the SessionStart hook indexes all synced memory files, so full recall is available from the first query. This is the same session-close discipline the guide already recommends; the two habits reinforce each other. After consolidating per-folder memories into the vault collection and re-indexing, cross-machine recall returns comprehensive history across folders. Zilliz Cloud (free tier, about 10 minutes) is the only configuration enabling a shared real-time index across machines, worth it only if you regularly run live sessions on two machines at once.

Index Maintenance

Mostly automatic: SessionStart re-indexes on open, the Stop hook on close. Three manual commands cover the edges: `memsearch stats` (confirm the index is populated when recall seems to miss obvious context), `memsearch compact "$MEMSEARCH_DIR/memory/"` (after roughly 60 daily files), and `memsearch reset --yes && memsearch index "$MEMSEARCH_DIR/memory/"` (after a machine migration, an embedding-model change, or noticeable quality drop). The Markdown files are never deleted by maintenance; the index always rebuilds from them.

Key Principles

1. **Install after the basic vault workflow is habitual.** A four-to-six-week lead time lets session history accumulate so recall has something to search.
2. **The Markdown files are the source of truth; the index is rebuildable.** A full reset and re-index restores everything from them.
3. **Common failures are setup-level, not operational.** Thin summaries usually mean a wrong or absent summarization model or an `ANTHROPIC_API_KEY` in the shell; empty recall usually means an unset `MEMSEARCH_DIR` or the wrong embedding model.

Appendix D: The Words People Use, a Concordance

Summary: The field has no agreed vocabulary. The same handful of terms, second brain, AI OS, PKM, life OS, agentic OS, work OS, name overlapping but different things depending on who is speaking. This appendix maps the terms you meet online to the vocabulary this guide uses, so a confusing video stops being confusing.

Why this appendix exists

Most people come to this through videos and blogs, and every influential voice invents or bends the words. Follow one voice and you get a partial map drawn for a different reader, often a software developer. Follow several and you end up in fog. This is the most common reason capable people stall: not the ideas, the words. The crosswalk below is the tool that clears it. When a source uses a term you are unsure of, find it here and read across.

This guide's vocabulary

The guide commits to a small set of terms and translates everything else into them.

- **Work OS.** The whole estate: your vault, its folders, and the context files the agent reads. The system this guide builds.
- **Folder (workspace).** The unit of work. One folder per bounded effort, each a pre-briefed specialist.
- **Context files.** `CLAUDE.md` (standing rules), the background file (running state), the working brief (current operating state).
- **Knowledge layer and action layer.** The two lenses on the one estate: what it remembers, and what it does (Chapter 4).
- **Knowledge base (the wiki).** One use case of the work OS: a folder whose job is to accumulate knowledge and surface it as a wiki. A room, not the house.
- **Harness.** The tool you run the agent through (Claude Code). How you operate the OS, not the OS itself.
- **Memory layer.** The automatic cross-session capture and recall beneath the curated files (Chapter 9, MemSearch).

Three role words: collaborator, agent, tool

One more piece of vocabulary discipline, this time about the AI itself. Three roles recur in this field's writing, and naming the role rather than the occupant keeps your own files readable when the product changes:

- **AI collaborator**: the AI working with you in session, synchronously, with you reading the output.
- **Agent**: AI running unattended, with nobody reading each step.
- **Tool**: deterministic machinery. Folders, scripts, hooks, recorders: things that do the same thing every time.

Never call the collaborator "the AI tool"; it files the judgment you work with under the word for machinery. This guide itself uses **agent** in a broader, older sense, the category on the far side of the chatbot (Chapter 3): software that reads and writes your files rather than answering in a sealed window. Most of what this guide describes is that agent working as a collaborator, with you in the room. Where the unattended sense is the point, the daily-briefing routine in Chapter 11 or the third operator position in Chapter 8's context-rot section, the text says so.

The crosswalk

Term as it appears online	What it usually means	Who uses it, and the context	Where it lands in this guide
Second brain	An external system to capture, organize, and retrieve your own knowledge; the human is the actor	Tiago Forte; the wider PKM world; the Obsidian-plus-AI video wave	A use case of the work OS: the knowledge base (Chapters 7 and 11). The human-actor version.
Personal Knowledge Management (PKM)	The discipline of organizing and retrieving your own knowledge	Broad, predates AI	The discipline underneath the knowledge base, not the whole OS (Chapter 7).
AI OS / AI Operating System	The system you "work out of" that acts for you	AI creators; developer projects	The work OS itself. Same idea, this guide's preferred name.
Agentic OS / Agentic AI OS	Agents, memory, tools, and workflows working as one coordinated whole	Vendor and platform writing	The action layer of the work OS, described as a stack (Chapter 4). A developer-centric framing.
Life OS / Agentic Life OS	A productized, all-in-one personal operating system	Notion, Tana, and similar product marketing	A marketing wrapper over the same idea; the guide avoids the term.
[Name]OS / myOS / personal	A personalized operating system	Creator and influencer	The work OS, personalized. A

Term as it appears online	What it usually means	Who uses it, and the context	Where it lands in this guide
OS	named for its owner	branding	naming style, not a different system.
Work OS	A system you run your work from	This guide; also Monday.com's product category and WorkOS, a developer-auth company	This guide's term for the whole estate. Note the name is already contested in the market.

A note carried from Chapter 4: where a term comes from a software-development source (agentic-OS stacks, testing tooling, project READMEs that call themselves an "operating system"), it was written for a builder, not for a knowledge worker. The idea usually transfers; the framing and the setup do not. Translate it into the vocabulary above and ignore the developer scaffolding.

Key Principles

1. **The words are the trap, not the ideas.** When a source confuses you, suspect the vocabulary first.
2. **Translate to one vocabulary.** Map every term to this guide's set and the contradictions between creators mostly dissolve.
3. **Second brain is a room.** Whenever you hear it, read it as the knowledge-accumulation use case, not the whole system.
4. **Mind the developer lens.** Much of the best-known content is written for builders; take the idea, drop the scaffolding.

Appendix E: The Doctrine Sheet

This appendix is the guide at a glance and the core an AI collaborator works from, in one place: a mechanical extraction of every chapter's section summaries and key principles, in reading order, regenerated from the guide's own text with every edition, so it can never drift from the book it distills. It serves two readers. For you, it is the revision surface: the whole system on a few pages, for the second read and the quick check. For an AI collaborator, it is the doctrine sheet the builder prompt in Chapter 11 points at: attach this PDF, direct the collaborator here, and it can apply the guide's rules without re-digesting every page. The reasoning, the examples, and the setup procedures live in the chapters.

Chapter 2 · The Reader and the Problem

Who This Is For

This guide is for the knowledge professional, in any size of organization, who wants agentic AI to grow their capacity, efficiency, and effectiveness, not to write software.

The Stateless Problem

Every AI session starts from zero by design, and that architectural fact becomes a serious friction point the moment AI is applied to recurring, evolving knowledge work.

Two Clarifications That Shape the System

Two boundary-setting clarifications before the disciplines: the system is folders and files, not any particular editor, and direct integrations are a later layer, not a starting requirement.

Capturing and Supplying Context

One discipline with two sides, capturing the implied context of your work into durable files and supplying it to the AI at the right moments, turns what only you know into a portable, file-resident system that any AI can consume from the first message.

Signal Over Noise

The discipline that governs the folder's context files, the files the agent loads to orient itself: each holds only what is relevant and durable at its level, and nothing else. It rules what the

agent reads, not what the folder may contain.

Chapter 3 · From Chatbots to Agents

The shift from an AI chatbot to an AI agent is a change of category, not degree: an agent reads and writes your files, uses tools, and persists context, which is what makes a durable work OS possible at all. The chapter also sets the frame the rest of the guide builds on: Karpathy's three layers, the spec, the verifier, and the environment.

Chapter 4 · One Estate, Two Lenses

The work OS is one estate, the vault, its folders, and the context files, seen through two lenses: what it remembers and what it does. The "second brain" the internet sells is one room in that estate, not the estate itself. Getting the words straight here is what stops you building the wrong system.

Key principles:

1. **One estate, two lenses.** The vault, its folders, and its context files are one system. "Remembers" and "acts" are two ways of looking at it, not two things to own.
2. **The knowledge layer comes first.** The action layer is built on it and is worthless without it. Build the memory before you automate.
3. **The second brain is a room.** It is the knowledge-accumulation use case, one application of the work OS, not the work OS itself. Do not mistake the room for the house.
4. **Let the lens pick the discipline.** Ask whether a file is to be remembered or acted on; the answer tells you whether to guard it with freshness or with restraint.
5. **When the words confuse you, go to the crosswalk.** Appendix D maps the popular terms to this guide's vocabulary.

Chapter 5 · Getting Set Up

An operating system starts with a habit, not an architecture: route everything through one agent and let the context accumulate. This chapter covers the one early choice you do need to make, where to run the agent, and a small input habit, speech-to-text, that changes how the tool feels to use.

Key principles:

1. **Default first, architecture later.** Route work through one agent in one place before you design any folder structure. The structure follows the habit.
2. **The environment is comfort, not capability, and it evolves.** The same engine and files run everywhere, so pick the surface you are happiest in. Most readers start in the desktop app and graduate to the CLI in a terminal as their work deepens; Warp is the terminal to land on.
3. **Lower the cost of giving context.** Dictation (Spokenly or SuperWhisper) makes full briefing cheap, which is what the rest of the guide depends on.
4. **Start with what is in front of you.** Do not let setup become the project. Begin, then improve only when friction appears.

Chapter 6 · Claude Code: The Agent of Choice

Claude Code Fundamentals

Claude Code is the agent this guide commits to: full read/write access to your local files, persistent context through plain Markdown, available as a desktop app for non-technical use and a CLI for developers, both reading the same files.

Key principles:

1. **Setup is three steps.** Open a session pointed at your project folder, from the desktop app or by typing `claude` in a terminal there, and your context loads automatically: if a `CLAUDE.md` exists in that folder or any parent, Claude Code reads it before your first word.
2. **Two doors, one engine.** Claude Code has a desktop app and a CLI accessed by typing `claude` in a terminal. The engine, the files, and the conventions are identical: `CLAUDE.md`, `-background.md`, slash commands, the `@file` syntax, the directory-tree walk at session start. Most online content shows the CLI because it is older and the default for technical users. Read past the interface details; the files and conventions transfer cleanly. The desktop app is the gentler starting door; most maturing users end up in the terminal, as Chapter 5 described, and both doors share the same `~/.claude/` directory, so nothing is lost in the move.
3. **Markdown is the format behind everything.** `CLAUDE.md`, `-background.md`, and reference files are all plain-text Markdown. A `#` is a heading, a `-` is a bullet, `**word**` is bold. Markdown is the standard across AI tools, carries no proprietary lock-in, and makes every file you write a portable knowledge asset. You do not need to learn the syntax first; the agent drafts the files from plain-language instructions.

4. **Persistence, not intelligence, is the upgrade.** The model is the same across modes. What changes is that the folder remembers, so the next session is productive from the first message.

The CLAUDE.md System

`CLAUDE.md` is the standing-rules file the agent reads automatically at every session start, and it lives at three nested layers, machine, vault, and folder, each doing one job.

Key principles:

1. **Let the agent draft the first file.** Setting up a new folder is the one moment a blank page is genuinely useful. Describe the folder's purpose and the standing rules you want, and ask for a draft `CLAUDE.md` and `-background.md`, applying signal/noise. You supply the substance; the agent supplies the structure. In a mature vault this is what the folder-init skill automates.
2. **The machine layer is machine-local.** `~/.claude/CLAUDE.md` is not carried by vault sync. Across machines, either sync `~/.claude/` via a dotfiles repository or maintain a parallel copy and reconcile. The same caveat applies to auto-memory.
3. **Apply the six-month test.** Anything still true in six months belongs in `CLAUDE.md`; anything that changes belongs in the background file. See Signal Over Noise in Chapter 2.
4. **Groom event-driven, not session-driven.** Update `CLAUDE.md` only when scope or a standing rule materially changed, never just because a session happened. A clean file beats a comprehensive one. Review all folder `CLAUDE.md` files quarterly.
5. **Context isolation by folder is intentional.** A session launched from one folder does not carry another folder's context. Keep each `CLAUDE.md` scoped strictly to its own folder; use `/add-dir` only when a task explicitly needs cross-folder context.

The Background File

The `-background.md` file holds a folder's running state, and it works through a load/update pair, an automatic read at session start and a deliberate write at session close, where the write side is the part that quietly fails.

Key principles:

1. **Update after material change, not after every session.** Keep the file structured (decisions, open items, current state) so it stays readable.

2. **The naming is convention, not standard.** Online content often calls the running-state file `MEMORY.md`; it is the same concept. Only `CLAUDE.md` is native to Claude Code. Pick one name and stay consistent.
3. **Do not confuse it with the agent's auto-memory.** A separate store, also indexed by a file called `MEMORY.md`, lives under `~/.claude/` and is written by the agent. See Auto-Memory in Chapter 9.
4. **Graduate the write side deliberately.** Start with the manual closing prompt. Move to the session-handoff skill once you know what a good close looks like. Use a Stop hook only at high session volume.

The Working Brief

When a folder shifts from knowledge storage to active operations, a third file, `[FolderName]-workingBrief.md`, separates durable memory from current operational state; the background file is the memory, the working brief is the steering wheel.

Key principles:

1. **Add a second eager-load instruction.** Update `CLAUDE.md` so both files load at session start: "read [FolderName]-background.md and [FolderName]-workingBrief.md before responding."
2. **Session-close routing for three-file folders.** For each item: does it answer "what became part of the lasting story?" (background file) or "what needs attention now?" (working brief)? If both, write it in different forms: the settled decision to the background file, the open action that follows to the working brief.
3. **The working brief is full-replace and always current.** Unlike the background file, which you update when something becomes permanent, the working brief should always reflect the present. A working brief two sessions out of date is worse than none: it misdirects the next session rather than orienting it. Carry a `Last updated: YYYY-MM-DD HH:MM` line.
4. **Record the previous working-brief state before overwriting.** Capture it as a dated summary in the background file first, so the steering history is preserved in the memory layer. The session-handoff skill does this automatically.
5. **Keep it short and scannable.** Its value is in being current, not comprehensive. Comprehensive is the background file's job.

How Standing Rules Fire: Reflex Rules and Hooks

Two different mechanisms make the system act on its own. A reflex rule is an instruction in `CLAUDE.md` the agent applies with judgment, and it is how a skill "auto-triggers." A hook is a command the harness runs on a mechanical event, deterministically, and it is how you enforce a rule that must never be broken. Knowing which is which is most of what it takes to automate well.

Key principles:

1. **Mechanical event, use a hook. Judgment about meaning, write a reflex rule.** This one question resolves almost every "how do I automate this" decision.
2. **Skills auto-trigger through reflex rules, not hooks.** The standing instruction in `CLAUDE.md` is what makes the agent reach for a skill at the right moment.
3. **Enforce never-do rules with hooks.** A `CLAUDE.md` line is a request; only a hook makes a rule the model cannot break.

Where a Rule Lives: CLAUDE.md or the Skill

Every reader who builds a second skill hits the same question: a new rule exists, so does it go in `CLAUDE.md` or in the skill? One test answers it: if the rule must fire before the agent knows it needs the skill, it is a trigger and belongs in `CLAUDE.md`; if it only executes inside the procedure, it belongs in the skill. The economics enforce the test: every `CLAUDE.md` line is paid for on every turn of every session, while a skill's body costs nothing until it fires.

Key principles:

1. **Triggers, invariants, and prohibitions live in `CLAUDE.md`; procedure lives in the skill.** If the rule must fire before the agent knows it needs the skill, it cannot live inside the skill.
2. **`CLAUDE.md` is the most expensive space in the system.** Every line is paid for on every turn of every session; a skill's body is free until it fires.
3. **Never restate the skill's steps in the trigger.** The rule says the skill must run; the skill owns what happens next.

Chapter 7 · The Folder as Workspace

Work OS vs Knowledge Base

Two systems are routinely conflated under labels like “second brain” and “work OS”: the work OS is a workspace for active work, the knowledge base is a library that compounds topic knowledge. This guide builds the work OS; the library is an applied system in its own right.

Key principles:

1. **Decide which system a folder is before you build it.** Active effort with stakeholders and deliverables is work OS. A compounding topic library is a knowledge base.
2. **PKM is the discipline, not the goal.** Capture, triage, curate. Do not build a second brain for its own sake; that is the wrong investment for this reader.
3. **Route by intention, not by content type.** The same domain fact goes to the knowledge base when captured to compound, and to the work folder when it serves the active effort. The two systems still do not merge.
4. **From here, the guide builds the work OS.** The library pattern returns in Chapter 11 as an applied system.

The Folder as the Organizing Artifact

The folder is the central artifact of the work OS: each folder is a pre-briefed specialist scoped to one area of work, folders are either dedicated or universal, they sit flat at the vault root, and the root hands its context down to every one of them automatically.

Key principles:

1. **One folder per bounded effort, flat at the root.** The folder is the workspace; scope it to one area of work and give it a place in the row, not a nest. Subfolders organize a workspace’s papers; they do not create workspaces.
2. **Adding a folder requires zero edits elsewhere** beyond a one-line vault-map entry. Drop in a `CLAUDE.md` and a `-background.md` and the folder is discoverable.
3. **Write it at the root once; every folder inherits it.** A folder’s `CLAUDE.md` adds what is specific to its effort; it never restates the root.
4. **Route by the vault map, not by listing the root.** The map is the durable index; keep it current.
5. **Each file written once is context you never re-explain.**

Cross-Folder Context

Two commands extend a session beyond its launch folder: `/add-dir` grants read access to a sibling folder, and `@file` pulls specific files into context on demand, with isolation as the default and access as a deliberate act.

Key principles:

1. **Use `@file` and `/add-dir`, never copy a file between folders to "make it available."** That is the entire purpose of this setup.
2. **Isolation is a feature.** Explicit scoping each session keeps unrelated context from leaking in.
3. **Repeated reaching is a redesign signal.** A file this folder always needs belongs here, in a universal folder, or behind a declared dependency; never leave it a habit.

Two Kinds of Vaults

Before adopting any cross-folder mechanism, classify your vault: in a one-principal vault (the executive's vault), context crossing folders compounds value; in a walled vault (the lawyer's vault), the same crossing is a breach. Every cross-folder pattern in this guide carries a label naming the kind it serves.

Shared Entity Context

A `_context/` layer holds reusable analytical profiles of recurring entities, and active entity matching at session start connects every folder to it through one maintained index rather than brittle per-folder reference lists.

Key principles:

1. **Keep explicit lists only for core entities.** For the two or three entities central to every session in a folder, an explicit entry loads them as part of session-start instructions, ahead of the matching step. Rely on auto-resolution for the long tail. Do not maintain both exhaustively.
2. **This is the horizontal axis of cross-folder awareness, and it is opt-in.** Root inheritance is the vertical axis (vault root to folder via `CLAUDE.md` traversal, tool behavior every vault gets); entity auto-resolution is the horizontal axis (vault-wide shared intelligence, a structure only the one-principal vault should build).
3. **`_context` files are stable reference intelligence, not operational state.** They are the one place where cross-linking between files earns its keep, because the files are stable and benefit from lateral navigation.

Chapter 8 · Operating Day to Day

Starting a Folder: folder-init

One question comes before a new folder exists: is this certainly its own folder, or is there doubt that its scope already lives somewhere in the vault? When it is certain, create the folder and start inside it; the root is never opened. Only when create-or-extend is in doubt is the folder born from the vault root, where the concierge start decides, names, and seeds it. Either way `folder-init`, the session-start counterpart to session-handoff, orients the new folder, first as one manual prompt and later as a skill of your own: it checks the vault map for duplicate scope, challenges the name, scans existing artifacts, interviews you only about what they do not answer, and builds the context files and a source inventory before any real work begins.

Key principles:

1. **Orient first, then build.** Scanning the artifacts before interviewing means you answer fewer questions and the drafts are grounded in evidence.
2. **Mark every inference.** Anything taken from artifact scanning rather than your direct statement is flagged for you to confirm or correct.
3. **Warrant the working brief; do not default to it.** It is created only when the folder shows multiple live workstreams, active coordination, or blocking decisions.
4. **Make the gap visible from day one.** The source inventory always exists, even empty, with a missing-context note.
5. **Whole-estate sight is a property of the vault map, not of the root session.** When a folder is certainly new, create it and start inside it; `folder-init` checks the map from there, and the root is never opened. Only when create-or-extend is genuinely in doubt is the birth root work, because that decision is about the estate, and the concierge start delivers it.

Ticket, Not Prompt: The Handoff Is the Bottleneck

As your use of AI matures, the scarce resource stops being the model and becomes the handoff: the moment work leaves one worker, human or AI, and must reach the next. The fix is to hand over work orders, not prompts, and most of the work order’s parts are rules this guide has already taught.

Key principles:

1. **A prompt asks for an answer; a ticket asks for a result.** Hand over statements of work, not questions.
2. **You already own the parts.** Stop conditions, receipts, and context-as-files are standing rules of this system; the work order puts them in one shape.
3. **Adopt by fit test, not by faith.** Take the discipline, defer the queue until unattended work arrives, refuse the external silo.

Plan Mode and Compact

In-session controls for high-stakes work: gate execution on an accepted plan (plan mode is that gate made mechanical), watch context utilization rather than the clock or an exchange count, take a clean exit at the ceiling, and keep `/compact` for the mid-flight case. Prevent rot first, detect it second.

Key principles:

1. **Gate execution on an accepted plan.** Context, then goal, then plan, then approval, then release; plan mode is the same gate enforced mechanically.
2. **Watch utilization, not exchange counts.** Keep the window's usage visible, exit near half at a natural break, and keep `/compact` for the mid-flight case.
3. **Compact loses the "how," keeps the "what."** Save the reasoning trail to a file first when it matters.
4. **Prevent rot first, detect it second.** A context ceiling and your own reading beat a canary while you are watching; the canary belongs where nobody is reading.

Session Continuity and Handoff

Session close is the write side of the load/update pair applied to a whole session: learn it first as a copy-ready closing prompt that writes the active folder's files, then mature it into the one-command `session-handoff` skill, so any later session, on any machine, opens fully oriented. How far beyond the folder a close should ever reach is a design choice, treated on its own below.

Key principles:

1. **The close is part of the session, not an add-on.** Skipping it means the next session starts one session stale, and staleness compounds.
2. **Durable to the background file, current to the working brief, never the reverse.** Dated sections keep the background a coherent narrative; the working brief stays a full-replace snapshot.

3. **The close always lands in the active folder; any reach beyond it is a setting you chose.** Default to the island close, and widen the reach only in a one-principal vault, along rules you wrote down.
4. **Groom standing rules and auto-memory on a slower cadence.** Update `CLAUDE.md` only on a real rule change (the six-month test); quarterly, audit auto-memory for entries that drifted or went stale.
5. **The transcript is the backstop, not the system.** A bad exit costs only the unrun handoff; resume the session, recover, then close properly.

Chapter 9 · The Memory Layer

Auto-Memory

Auto-memory is a third memory mechanism, an autonomous store the agent writes on its own initiative under `~/.claude/`, useful for durable cross-folder facts about you and unhelpful for anything folder-specific.

Key principles:

1. **Keep it only until MemSearch is in place.** On a single local project the native store is a fair convenience. Once you run MemSearch at the vault root it becomes a second, hidden, machine-local copy of your context, and the closing section of this chapter (Two Memory Layers, One to Keep) explains why to turn it off. Until then, audit it quarterly: ask the agent to list everything per active folder and prune.
2. **Redirect folder-specific offers.** When the agent offers to save something, ask whether it applies across folders or only inside this one. If the latter, send it to the folder's `CLAUDE.md` or `-background.md`.
3. **Auto-memory is not portable.** It lives outside the vault, so vault sync does not carry it; a second machine starts with an empty store. This is the single largest departure from the principle that everything important is a file you can see and edit, which is why the system's center of gravity stays in the vault files.
4. **Default to declining while learning.** While building the discipline of authoring your own files, decline the save offers. Auto-memory becomes a useful auxiliary later, not a replacement for the file-resident system.

MemSearch: the Memory Layer

MemSearch sits beneath the curated context system, automatically capturing every session and making it findable through vault-wide semantic recall; configured at the vault root, it searches across every folder, and it amplifies context engineering rather than replacing it.

Key principles:

1. **The one new reflex.** When uncertain about past context, reach for `/memory-recall` before hunting through files. Outside recall moments, MemSearch is invisible.
2. **Evaluate over six to eight weeks, not one.** The index is thin early; value compounds with accumulated history. Install once the curated workflow is already habitual.
3. **The clean session-close doubles as the multi-machine optimum.** Closing one machine before opening another lets the next session index the synced memory files and recall fully from the first query.
4. **The Markdown files are the source of truth.** The index is always rebuildable from them; SessionStart re-indexes on open and the Stop hook re-indexes on close, so maintenance is mostly automatic.

Two Memory Layers, One to Keep

Once MemSearch runs at the vault root, you have two memory systems with the same name doing different jobs: the native auto-memory store and MemSearch. For a multi-folder vault synced across machines, only one earns its place. Turn the native store off, on every machine, and let the vault hold the memory.

Key principles:

1. **Keep one memory layer, the one that serves the vault.** Once MemSearch is at the vault root, the native store is redundant and unsyncable. Turn it off.
2. **Turn it off on every machine.** The settings file and shell profile are machine-local; a change on one machine does not carry to the other.
3. **Let the vault hold the memory.** Durable context belongs in files you can see, edit, and sync, not in a private store under `~/.claude`.

Chapter 10 · Connecting Live Context Sources

Beyond the files you author, the rest of your work context lives in your communication channels, email, calendar, chat, and meetings. Connectors let the agent read the live ones

directly, and a call transcript can simply be dropped into the folder as a file. Either way, the copy-paste moment disappears.

Key principles:

1. **Map your context origins.** Context comes from you and from your communication channels. Know which channels carry your real working record, usually email, calendar, chat, and meetings, and bring those in first.
2. **A connector is a means to an end.** You care about the outcome, the agent reads the source, not the mechanism. Judge a connector by whether it removes a copy-paste step you do by hand today.
3. **The easiest source is often a file already in the folder.** A call transcript needs no connector; drop it in and reference it. Not all live context requires a live connection.
4. **Channel context feeds the files; it does not replace them.** Bring it in, then let the close curate the durable parts into the work OS.
5. **Scope connectors deliberately.** Active only where relevant, disconnected elsewhere, never crossed between accounts or workspaces.

Chapter 11 · Applied Systems

Building a Knowledge Base Wiki

The companion to the work OS is the knowledge base: a topic library, built on the Karpathy LLM Wiki pattern, where you drop raw sources into `raw/`, the agent maintains a cross-referenced `wiki/`, and answers saved to `outputs/` feed back to compound the knowledge over time.

Key principles:

1. **The agent is the librarian; never edit `wiki/` by hand.** You source and curate; the agent writes and maintains.
2. **`raw/` is immutable.** Drop sources in without organizing; never reorganize or delete by hand.
3. **Close the loop.** Outputs with durable insight go back into `raw/` so the next ingest compounds them.
4. **Lint on a cadence.** Monthly health checks keep contradictions, orphans, and stale claims from accumulating.

5. **Keep it separate from the work OS.** Topic knowledge routes here; project state stays in the work-OS folder.

A Publishing Pipeline

A four-queue pipeline where an article's state is its folder location, its metadata travels inside the file, side work arrives as work orders, and two firewalls, career and confidentiality, gate what may publish at all; the board that reports it is a generated view, never hand-maintained.

Daily Briefing Agent

A morning brief you can run today with one prompt, and the scheduled agent it grows into: business context pulled from your channels into the vault on a cadence, applying session-handoff routing in reverse to complete the in-out cycle of the work OS.

Key principles:

1. **Run the brief manually first.** The Phase 0 prompt delivers most of the value today and becomes the specification for the scheduled agent; automate on repetition, not on ambition.
2. **The vault structure is the classifier.** Reuse the close's routing rather than inventing a parallel scheme.
3. **Prove the intelligence layer before the display layer.** The dashboard depends on the agent working.
4. **Start with the immediately accessible source.** One email account validates the architecture; the rest follow once access is resolved.

Learning Resources

For the knowledge worker, creator selection matters more than video selection; three or four channels cover the full spectrum, and this guide itself can be a source document for an AI-generated learning track.

Key principles:

1. **Curate by creator background, not by video.** Channels stay accurate longer than individual recommendations.
2. **Prioritize mental models over features.** Workflow principles compound; feature tours age.

3. **The onboarding sequence is a worked example of audience-fit curation.** Build learning paths the same way.

Chapter 12 · How the System Grows

The system is never finished; it matures through use. Complexity is added only when a real session exposes a gap, and only in the lowest-friction way that closes it. This is the one idea to keep after the mechanics fade.

Appendix A · Worked Example

One concrete pass through the matured system end to end: open a folder with folder-init, hand the work over as a work order gated by plan mode, keep the session inside its context ceiling, then close with session-handoff, against a vault whose root files and vault map orient every session and whose MemSearch layer captures it all.

Appendix B · Environment, Setup, and Browser Tools

`settings.json` offers two levers for persistent cross-folder context, auto-loading sibling `CLAUDE.md` files and auto-granting folder access, but the default workflow needs neither; add persistence only when a session pattern repeats often enough that typing `/add-dir` becomes real friction.

Key principles:

1. **Start with no settings.** Work the default flow for a few weeks and let friction surface organically.
2. **Prefer `settings.local.json` and folder-scope over user-wide.** The `.local` variant is excluded from version control, and scoped beats global, mirroring signal over noise. Keep `~/.claude/settings.json` minimal.
3. **Document persistence inline.** JSON has no comments, so add a note at the top of the folder's `CLAUDE.md` explaining why siblings are pre-loaded.
4. **Re-evaluate quarterly.** Persistent settings drift; folders used constantly six months ago may be cold now.
5. **Avoid persistence for sensitive folders, large or binary content, and short-lived projects.** Verify active settings with `/status`, which also flags malformed JSON (the most common failure).

Appendix C · MemSearch Installation

A no-coding setup for MemSearch on macOS: install the CLI and plugin, configure local ONNX embeddings and Sonnet summarization, set `MEMSEARCH_DIR` to the vault root for vault-wide recall, and verify with a fresh-session capture and a cross-folder recall test.

Key principles:

1. **Install after the basic vault workflow is habitual.** A four-to-six-week lead time lets session history accumulate so recall has something to search.
2. **The Markdown files are the source of truth; the index is rebuildable.** A full reset and re-index restores everything from them.
3. **Common failures are setup-level, not operational.** Thin summaries usually mean a wrong or absent summarization model or an `ANTHROPIC_API_KEY` in the shell; empty recall usually means an unset `MEMSEARCH_DIR` or the wrong embedding model.

Appendix D · The Words People Use, a Concordance

The field has no agreed vocabulary. The same handful of terms, second brain, AI OS, PKM, life OS, agentic OS, work OS, name overlapping but different things depending on who is speaking. This appendix maps the terms you meet online to the vocabulary this guide uses, so a confusing video stops being confusing.

Key principles:

1. **The words are the trap, not the ideas.** When a source confuses you, suspect the vocabulary first.
2. **Translate to one vocabulary.** Map every term to this guide's set and the contradictions between creators mostly dissolve.
3. **Second brain is a room.** Whenever you hear it, read it as the knowledge-accumulation use case, not the whole system.
4. **Mind the developer lens.** Much of the best-known content is written for builders; take the idea, drop the scaffolding.