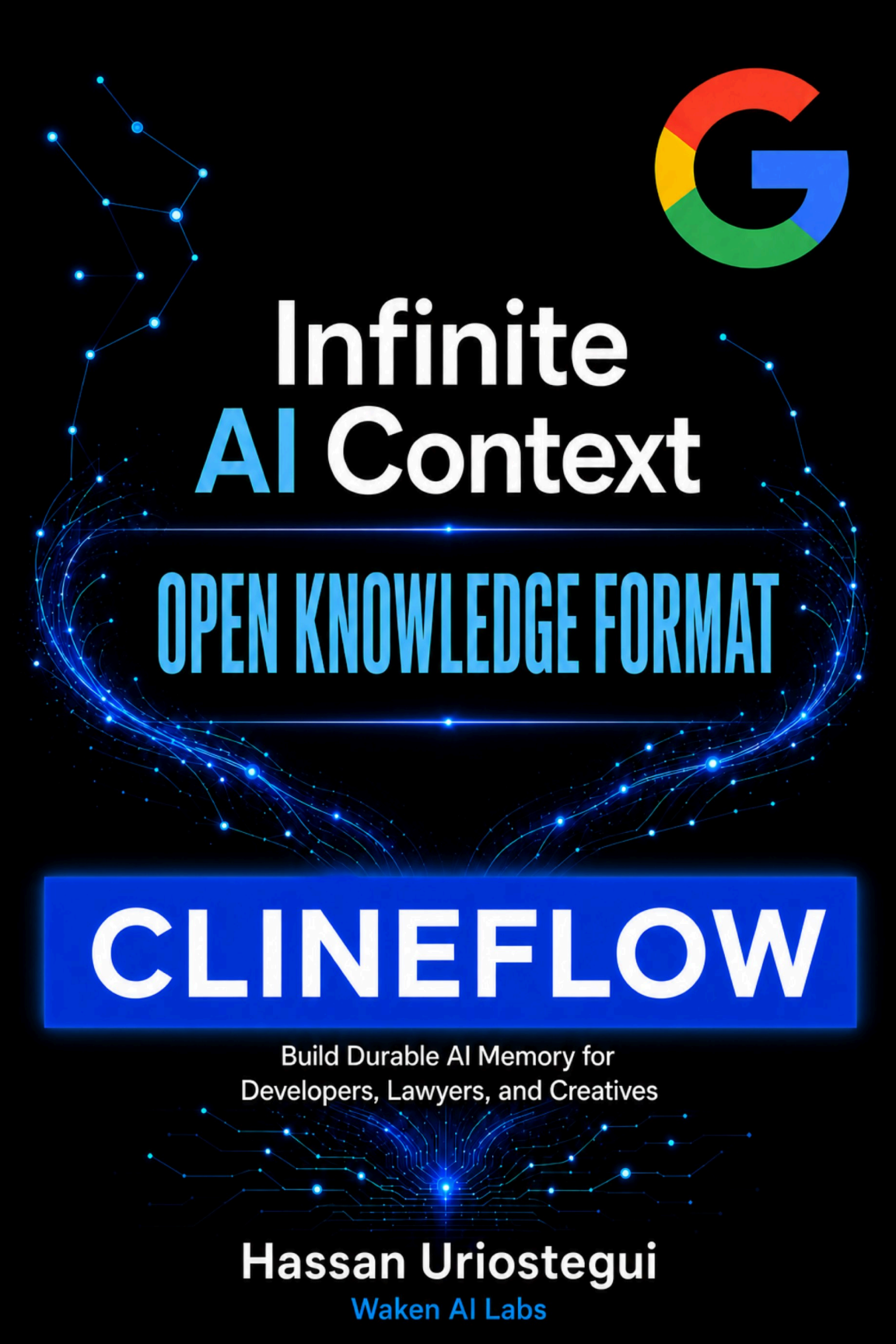

Infinite
AI Context
OPEN KNOWLEDGE FORMAT
CLINEFLOW
Build Durable AI Memory for
Developers, Lawyers, and Creatives
Hassan Uriostegui
Waken AI Labs

Infinite AI Context: ClineFlow and Google’s Open Knowledge Format

Build Durable AI Memory for Developers, Lawyers, and Creatives

Hassan Uriostegui

Imprint: Lulu.com

ISBN 978-0-557-93761-5

For Fernanda Beltran, my wife, and Aurora Cotne, my mother.

You have always supported me, believed in me, and moved me forward with your light and infinite love. May these pages help Fernanda create her stories and unleash her narrative genius; and may they support my mother, an attorney, in her legal practice.

About the Author

Hassan Uriostegui is a Mexican-American technologist, author, and founder of Waken AI Labs. His work sits at the intersection of AI, software development, creative tools, and human-centered technology.

Across a career spanning mobile visual effects, video software, and AI-human interaction, Hassan has helped build products including Viddy, Community, Ultrakam, and FlyrTV. He created ClineFlow to make the project knowledge behind an AI-assisted task durable, portable, and useful across tools and sessions.

His writing and product work advocate for AI that extends human agency: technology that makes it easier to preserve decisions, create responsibly, and keep people in control of their own context.

Acknowledgements

To my friend Toufeeq Hussain: thank you for your confidence in me and for opening the door to collaborate in the world of AI gaming. Our peer friendship, now more than a decade long, is built on curiosity, generosity, and a shared willingness to imagine what may be possible next.

How to Read This Book

You can read this book from cover to cover, but it is designed as a field guide: learn the memory system first, then apply it to the project that fits your work. Chapters 1–3 explain the foundation —Open Knowledge Format (OKF), ClineFlow, and a local or cloud AI setup. Chapters 4–9 are hands-on labs. Choose the lab closest to your goal: a finance dashboard, portfolio, web app, iOS app, fiction project, or synthetic legal training matter.

Every lab follows the same loop:

```
locate → understand → introspect → plan → define success → ground →
execute →
verify → journal → continue
```

Read the short outcome and safety boundary at the start of a chapter before you prompt an agent. Then copy the prompt sequence into your chosen AI tool, keep the task narrow, inspect what the agent proposes, run the stated verification, and record what actually happened in the project journal. The prompts are starting points, not commands to follow blindly.

What is online

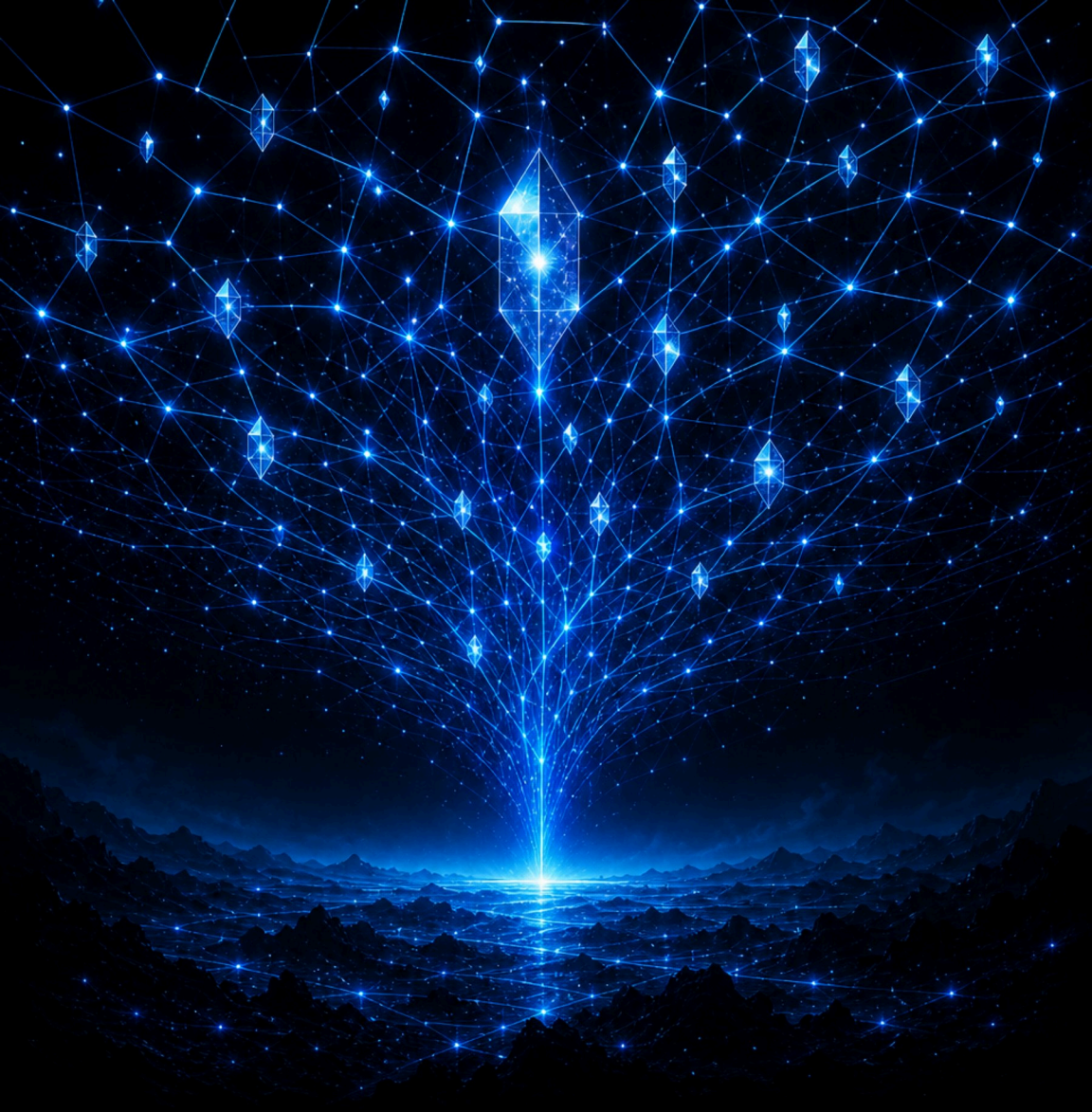
Each practical chapter has a reader-bridge card with a QR code and companion address. The companion is organized as one local project per chapter. It contains the starter material, completed example, prompts, checklists, and the knowledge records that show how context is retained. Scan the code with your phone camera, or type the printed address into a browser.

The companion is live at `https://hassanvfx.github.io/infinite-ai-context/`. The printed book remains complete on its own; the online material is an optional place to download the labs and find future updates. Do not upload private project material, credentials, client files, or personal data to any online service.

Choose your path

- **New to tools?** Read Chapters 1 and 2, then start with Chapter 8′s fiction project or Chapter 5′s portfolio exercise.
- **Developer?** Read Chapters 1–3, then choose Chapters 4, 6, or 7.
- **Legal professional?** Read Chapters 1–2, then Chapter 9. Use only the synthetic material supplied; the chapter is not legal advice and does not replace professional judgment.
- **Using local AI?** Complete Chapter 3 before a lab. If local hardware is not a good fit, use a cloud agent—the ClineFlow memory workflow is the same.

Contents



PART I

THE MEMORY LAYER

Build context that survives the session.

CHAPTER 01

OPEN KNOWLEDGE, DURABLE CONTEXT

Portable knowledge that survives the chat.

Reader outcome

You will understand why a project can have more useful memory than any single chat, and how Open Knowledge Format (OKF) turns everyday files into a portable knowledge layer.

The real bottleneck is context loss

An agent can write a useful first draft quickly. The harder question arrives tomorrow: can the next conversation recover what the team decided, what changed, which evidence was checked, and what still needs attention? When the answer is no, people repeat explanations and agents repeat investigation. The failure is not simply a short model context window. Important project knowledge has been left inside a temporary conversation instead of being made reviewable, findable, and durable.

This book uses **infinite AI context** as a practical metaphor. It does not mean that a model receives unlimited tokens. It means that the most valuable context can live outside the chat: in a repository the team can inspect, improve, version, and hand to another tool. A conversation may end. The project's working memory does not need to end with it.[1]

That distinction is liberating. You do not need to choose one permanent AI provider, one IDE, or one enormous prompt. You need a durable place for the information that should survive: the intended outcome, the current state, constraints, terminology, decisions, tests, risks, and next safe step.

[1] Hassan Uriostegui, "Adopt Google's Open Knowledge Format with Codex — or Any Copilot."

What OKF contributes

Google's Open Knowledge Format is deliberately small. An OKF bundle is a directory of ordinary Markdown files with YAML frontmatter. Each file is a concept. The path gives the concept an identity; the frontmatter supplies a small amount of queryable structure; the Markdown body explains the idea to a human; ordinary links connect related concepts.[2]

Here is the important part: this is not a new database, a required cloud service, or a model runtime. A bundle can live in a Git repository, a folder, or an archive. Someone can read it in a text editor. An agent can navigate it. A different tool can consume the same files later. That is portability in a form that is easy to inspect.

At its simplest, a project might look like this:

```
knowledge/
  index.md
  journals/
    dashboard.md
  decisions/
    local-data-only.md
  playbooks/
    release-check.md
```

`index.md` helps a reader or agent discover what exists before opening everything. A journal records a task's changing story. A decision record says why an important choice was made. A playbook captures a repeatable procedure. Links turn these files from a pile of notes into a navigable web of context.

This diagram shows the files ClineFlow creates and maintains for you after installation. It is not a homework assignment to build a Markdown folder by hand. You give the agent a goal, constraints, and approval; ClineFlow gives the agent the instructions and structure to create the task journal, capture decisions, record verification, and carry the next step forward. Your job is

[2] Google Open Knowledge Format introduction and v0.2 specification.

to judge whether the goal, decisions, and evidence are right—not to become the project's clerk.[3]

Think of this as a local, Git-native project database expressed in ordinary OKF files—not as another chat to maintain. It is transparent: you can inspect every record, while ClineFlow updates the context plumbing in the background. When a task is verified, say **"please commit."** The agent pairs the work with its updated journal and handoff. A new chat, a different IDE agent, or a cloud model can then start from the same local project memory instead of asking you to reconstruct the previous conversation.[4]

Trust is part of context

Useful context is not only descriptive; it also tells a future reader how much to trust it. OKF v0.2 makes room for provenance, verification, lifecycle, and attestation. A concept may record where it came from, who verified it, whether it is draft or deprecated, and when it should be reconsidered.[5]

For a beginner, this can be simple: tell ClineFlow the source, date, and reviewer whenever a claim matters, and let it preserve that provenance. For a lawyer, it can mean keeping an authoritative document and a review note separate from a generated summary. For a writer, it can mean asking the flow to mark a character fact as a draft decision rather than settled canon. For a developer, it can mean having the agent record which test actually passed instead of saying "it should work."

What OKF does not do

OKF does not make a model correct. It does not replace code review, professional judgment, a legal review, a database schema,

[3] ClineFlow repository comparison and workflow documentation.
[4] ClineFlow repository comparison and workflow documentation.
[5] Google Open Knowledge Format introduction and v0.2 specification.

or a retrieval system. It is a format for the context those systems need. With ClineFlow installed, the agent maintains the files and returns to them as the project changes; the human still checks that the recorded facts, decisions, and evidence are honest.

This also keeps the comparison with other approaches fair. A product such as Kiro can offer an integrated agent environment. A framework such as spec-kit can impose a more formal specification workflow. ClineFlow focuses on a different layer: persistent, Git-native task memory that can sit alongside the agent you already use. Those approaches can overlap; the right choice depends on how much ceremony, tooling change, and formal planning a project needs.[6]

Try the automatic path

Open a dedicated project folder and tell your agent: **“Please follow the installation guide at https://github.com/hassanvfx/clineflow and install ClineFlow.”** Then give it this first task:

```
Please follow the installation guide at https://github.com/hassanvfx/clineflow
and install ClineFlow. Help me begin this goal: [your goal]. Preserve these
constraints: [constraints]. Start a journal for this task, map the project,
and ask for approval before making product changes.
```

The agent creates the initial structure and journal. You do not need to type YAML or decide a folder taxonomy. Read its proposed goal, constraints, and open question; correct anything important before you authorize the next step.

Common failure modes

- Treating a journal as a diary instead of a decision tool. Ask ClineFlow to record why a choice matters and how it was checked.

[6] ClineFlow repository comparison and workflow documentation.

- Saving context but never indexing it. Let ClineFlow maintain a short entry point so an agent can discover the relevant files.
- Calling every generated note authoritative. Preserve sources and mark draft material honestly.
- Building one giant “memory” document. Split knowledge by task or concept and connect it with links.

A worked bundle: one project, many useful memories

Imagine that you are building the personal-finance dashboard from Chapter 4. A first conversation produces a rough screen and a list of expenses. A second conversation changes the monthly budget rule. A third discovers that a chart looks wrong when a category has no transactions. If the only record is the chat history, the next agent has to infer which version of the rule is current and whether the empty-state bug was actually tested.

An OKF-shaped bundle gives each kind of information a home:

```
knowledge/
  index.md
  journals/
    finance-dashboard.md
  decisions/
    monthly-budget-rule.md
  tests/
    empty-category-check.md
  references/
    sample-data-policy.md
```

The journal is the moving task record. It says what the team is trying to build, what it learned, and what happens next. The decision document is slower-moving: it says that a monthly budget is a planned limit rather than money already spent. The test note captures the observable check: a category with zero transactions displays a zero value and does not break the chart. The reference document explains that the application uses synthetic data and never imports a reader’s financial information.

These files do not need a clever database. Their strength is that they can be read together. The journal can link to the budget decision, the test note can link to the bug that motivated it, and the top-level index can point a new agent to the active task. That is progressive disclosure: start with a small map, open the relevant path, and avoid flooding a fresh conversation with every file in the repository.[7]

A small concept document

Here is a deliberately modest decision record:

```
---
type: Decision
title: Monthly budget rule
status: stable
sources:
  - id: product-brief
    resource: /references/product-brief.md
---

# Decision

The dashboard compares each category's month-to-date spending
with a user-defined planned limit. It does not predict income.

# Consequence

Empty categories show zero spending and retain their limit.
```

This document is useful because it distinguishes an intention from an implementation detail. A future agent can still ask questions, but it does not need to guess whether “budget” means a forecast, a bank balance, or a spending cap. The same pattern works outside software. A fiction project can use a decision record to mark a character’s age as canon, provisional, or replaced. A legal workflow can use it to distinguish a source document from an internal working summary. In every case, the user decides what the record means; the format makes the distinction visible.

[7] Google Open Knowledge Format introduction and v0.2 specification.

A comparison exercise, not a product contest

When evaluating ClineFlow, Kiro, spec-kit, or any newer tool, avoid the question “which one is best?” Start with the failure you want to prevent. If a team repeatedly loses reasoning between agent sessions, a journal-centered memory layer solves a direct problem. If the team needs formal requirements and implementation phases before it writes code, a specification-first workflow may be appropriate. If the team wants a single integrated environment, an AI-first IDE may be attractive.

ClineFlow’s central promise is narrower and therefore easier to test: keep durable context beside the project and make it available to the next human or compatible agent. It does not require local models; it does not require a particular IDE; and it does not eliminate the need to decide whether the saved context is correct.[8]

Try this comparison exercise. Write three sentences: the project fact that must survive a handoff, the person or agent that will need it next, and the evidence that would show it is still true. If your current workflow has nowhere clear to put those sentences, you have found the first useful job for an open knowledge bundle.

Field test: turn one messy request into durable context

You do not need a large codebase to practice this. Pick a small project that you already understand: a website update, an event plan, a story revision, a client intake template, or a household budget. Write the request as you would normally give it to an agent. It may be vague: “make the home page better” or “help me organize this case.” Then pause before asking the agent to change anything.

[8] ClineFlow repository comparison and workflow documentation.

Ask ClineFlow to bootstrap the smallest project memory with four entry points: **purpose**, **current state**, **decisions**, and **active work**. Tell the agent who the work is for, what a good result should enable, and the constraints it must not silently change. It writes the first files and links; you review the result as a map of reality, not as a form to complete.

This is not bureaucracy. It is a way to separate facts, choices, and guesses before an agent turns a guess into files. When a project has a clear current state, an agent can investigate it. When a project has a clear decision, an agent can preserve it. When an item is only an open question, the agent can help research it without pretending it is settled.

Now use this prompt:

```
Check the journal and help me understand the current project before changing
anything. Explain what is confirmed, what is still unknown, and the next safe
step. Update the journal with confirmed facts and open questions; show me that
update before proceeding.
```

Notice what this prompt does not say. It does not prescribe an implementation before the agent understands the terrain. It asks for a map first. That is the first part of the grounded workflow developed later in this book: locate, understand, introspect, plan, define success, ground, execute, verify, journal, and continue.[9]

When the agent replies, do a human check. Are the files or documents it selected actually relevant? Did it confuse an old draft with the current source of truth? Did it find an ambiguity you had not noticed? Correct the shared understanding now. The earlier you correct a mismatch, the less expensive it is to repair.

Next, add a success note. It should be observable. “Improve the dashboard” is an intention, not a test. “The dashboard shows planned versus actual monthly spending; empty categories display safely; sample data remains local; and the mobile layout remains readable” is a testable definition of success. A legal equivalent might be “the assistant produces a source-linked issue

[9] Hassan Uriostegui, “Adopt Google’s Open Knowledge Format with Codex — or Any Copilot.”

map for the fictional matter and labels every generated summary as a draft for human review." A fiction equivalent might be "the chapter revision preserves the protagonist's established motive and flags any new fact that conflicts with the story bible."

Finally, authorize one small change. ClineFlow updates the active journal with what changed, what was verified, and what should happen next. Review that automatic handoff, then continue or stop. The next session begins with an inspectable trail rather than someone remembering a perfect conversation.

Keep the bundle humane

The format is most useful when it is lighter than the problem it solves. A five-page folder structure for a ten-minute task is not a victory. Start with the smallest set of durable records that answers three questions: What are we trying to do? What do we know? What should happen next? Expand the bundle only when a new concept needs a stable home.

The same restraint protects non-developers. You do not need to write YAML by hand on day one. ClineFlow and other tools can create a basic structure. Your responsibility is to read what becomes important, correct it when it is wrong, and decide what may be treated as settled. The files are there to make your judgment easier to carry forward—not to replace it.

Handoff checklist and journal example

Before you leave a task—whether you are stopping for lunch or handing it to a colleague—use this short checklist:

- **Outcome:** What did the task intend to make possible?
- **Current state:** What actually changed? Name the files, records, or decisions.
- **Verification:** What did you inspect, run, or compare? What remains unverified?

- **Open risk:** What could make the result misleading, unsafe, or incomplete?
- **Next safe step:** What should the next person investigate before changing anything else?

This checklist does not require a developer's vocabulary. A writer can say, "The protagonist's timeline is consistent through Chapter 6; the father's age remains a deliberate open question." A designer can say, "The mobile navigation was reviewed at a narrow viewport; keyboard navigation still needs a check." A lawyer working with a synthetic training matter can say, "The issue list is a draft generated from supplied fictional documents and requires attorney review before reliance."

Here is a small journal entry for the finance dashboard:

```
---
type: Engineering Journal
title: Finance dashboard empty-state check
status: draft
---

# Goal

Keep the category chart readable when a category has no transactions.

# Confirmed

The chart receives an empty array for a zero-transaction category.

# Decision

Display a zero-spend card and a plain-language empty-state message.

# Verification

Loaded the synthetic \u201ctransport\u201d category with no transactions.
The page rendered without an exception and showed $0 spent.

# Next step

Check the same state at a narrow mobile width.
```

The value is not the exact headings. The value is the distinction between a goal, a confirmed fact, a decision, a check, and an open next step. When these are blended into one paragraph, a later reader cannot tell what they may rely on. When they are

separated, an agent can use the journal as a map and a human can challenge any part of it.

Build checkpoints

Use checkpoints to prevent a project from becoming a long, opaque agent run. At the **map checkpoint**, the agent identifies the relevant files and explains the current behavior. At the **plan checkpoint**, you agree on success criteria and constraints. At the **change checkpoint**, the agent makes a small, reviewable change. At the **verification checkpoint**, it reports the exact evidence: a test command, a visible result, a comparison, or a remaining limitation. At the **handoff checkpoint**, it updates the journal.

For a simple project, these checkpoints can happen in one sitting. For a larger project, they may be separate sessions. Either way, the journal makes each boundary explicit. This is particularly useful when local models are used: a smaller, focused task gives a constrained model less unrelated material to juggle, while the knowledge bundle preserves the details it needs on the next task.

A note on links and evidence

Do not add links merely to make a folder look sophisticated. A link should answer a real reader question: “Where did this fact come from?” “Which decision controls this behavior?” “What test proves the result?” The most valuable link is often the one that lets a skeptical reader trace a conclusion back to a source or a check.

If a source changes, do not silently rewrite history. Update the affected concept, note the date, and say what changed. If a fact is no longer current, mark it as stale or deprecated instead of leaving a future agent to discover the contradiction by accident. These are small habits, but they make an AI workflow easier to supervise.

Next step

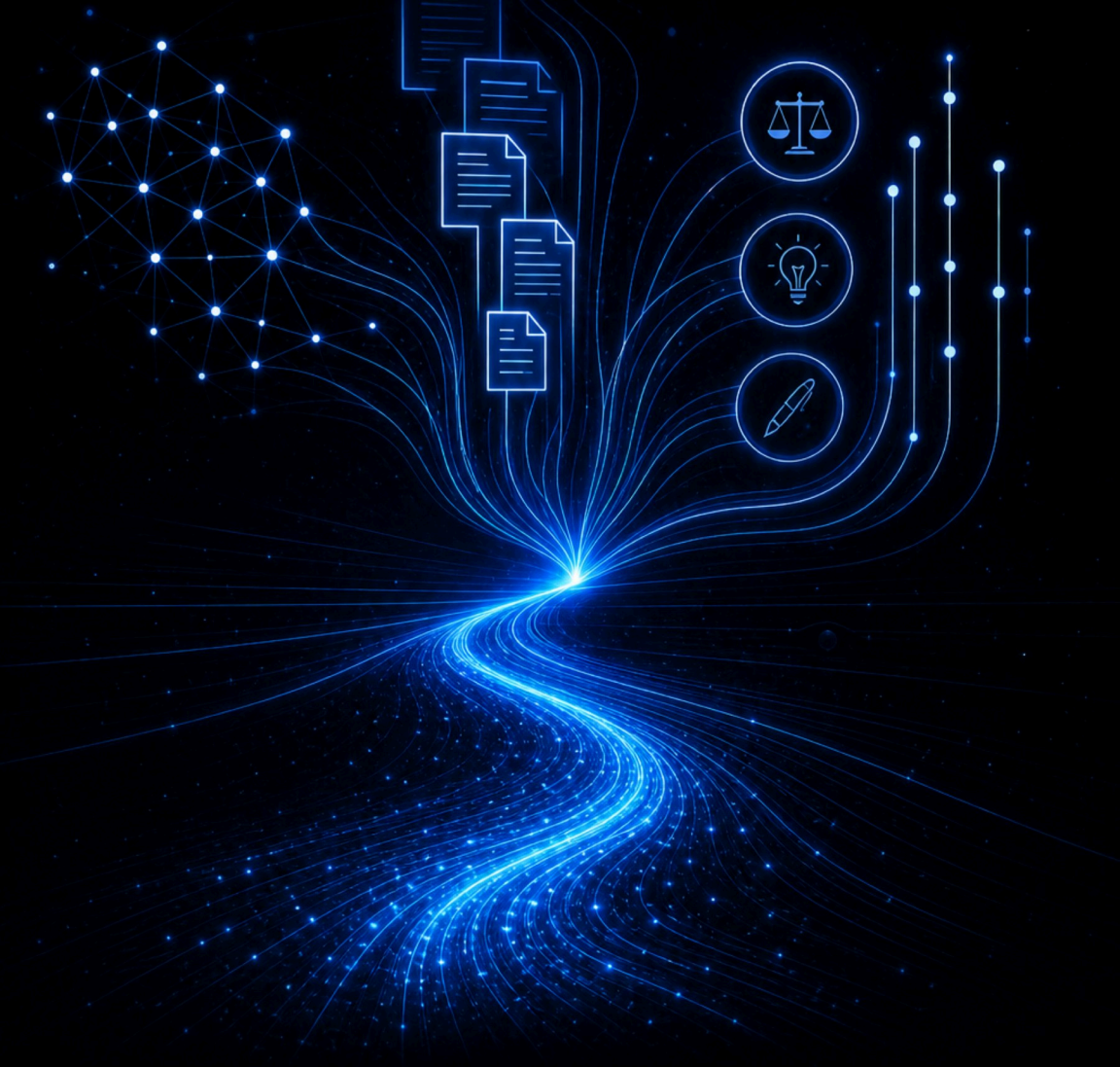
You now have the mental model: context is a maintained asset, not a giant prompt. Chapter 2 turns it into a practical ClineFlow setup. You will assemble a beginner toolset, install the workflow, inspect the generated knowledge bundle, and run the first health check. The goal is not to memorize commands. It is to give every later project a reliable place to begin.

Continue the chapter online

Map a portable knowledge bundle without turning unknowns into facts.

Next: Ask ClineFlow to check the journal and map the project.

Open the companion lab →



CHAPTER 02

CLINEFLOW: THE MEMORY LAYER

A durable record for every next step.

Reader outcome

Install ClineFlow, understand its knowledge bundle, and use its journal workflow with the AI agent you already prefer.

Planned sections

- The beginner toolset: Git, terminal, editor, and agent.
- Install and verify ClineFlow.
- Use the journal to start, resume, and hand off work naturally.
- The grounded prompting loop and Git-backed verification.

Start with a small toolset

You do not need to become a full-time developer to use ClineFlow well. You do need a few durable tools, because the system is designed to keep useful project knowledge beside the work rather than inside one web page.

Git is the history tool. Think of it as an undoable timeline for a folder. It can show what changed, compare one version to another, and let a team review context together with code or writing. You do not need to memorize every Git command before starting. The important habit is simpler: make a checkpoint after a meaningful, verified change instead of trusting a chat transcript as the only record.

A code editor is where you read and edit the project files. Visual Studio Code is a common choice, but any editor that can open a folder and display Markdown works for the knowledge layer. Writers and lawyers can begin by reading the Markdown files in an editor; the agent can help with the more technical parts as needed.

A terminal is a text-based way to run commands in the project folder. On a Mac, this is the Terminal application. Commands are useful because they make verification repeatable. A button labelled “worked” disappears with a particular product interface. A recorded command and its result can be checked again later.

An AI agent is the assistant that can read files, make planned changes, and run approved checks. ClineFlow is deliberately agent-agnostic: its repository documents configuration paths for Cline, ChatGPT Codex, Cursor, GitHub Copilot, Windsurf, and compatible future tools. The durable part is the project’s knowledge bundle, not the logo on the chat window.[10]

Finally, create a dedicated folder for each project. Do not point an agent at your entire home directory. Keep source material, generated work, and sensitive documents separated. For the examples in this book, use synthetic names and sample data. The legal assistant chapter is specifically a fictional training matter, never a place to upload real client material.

Install ClineFlow

The installation request is deliberately simple and should always use the official guide: **“Please follow the installation guide at https://github.com/hassanvfx/clineflow and install ClineFlow.”** If you prefer the terminal route, the same repository documents an installer command. Before running any downloaded script, read the repository, confirm the URL, and understand that it will modify the current project. Installation is a project action, not something to run casually in a folder containing unrelated files.[11]

After installation, the project should contain the shared instructions and the knowledge bundle. The exact layout can evolve, but the important pieces are familiar:

[10] ClineFlow repository and its documented agent-agnostic setup workflow.
[11] ClineFlow repository and its documented agent-agnostic setup workflow.

```
AGENTS.md                # shared project instructions
knowledge/
  index.md               # starting map for the knowledge bundle
  journals/              # task history, decisions, verification, next steps
clineflow-doctor         # local setup check
validate-okf             # structure check for the knowledge bundle
```

If the project already has an `AGENTS.md`, do not erase it. ClineFlow is designed to preserve existing project rules and add the shared workflow alongside them. That is a practical sign of the system's philosophy: durable context is an addition to a project's real conventions, not a reason to flatten them.[12]

Your first safe prompt

Once ClineFlow is installed, do not begin with "build my entire app." Begin with discovery in your normal language:

```
Check the journal and help me understand the current project. Summarize what is
already confirmed, identify any missing context, and propose the next safe
step. Do not change anything yet.
```

This prompt gives the agent a bounded job. It must read the map, look for prior work, and report uncertainty before it edits anything. If the repository is new, the honest answer may be that the project has no history yet. That is useful: it tells ClineFlow to create the first journal instead of pretending there is context to recover.

The journal is the only reader-facing memory tool

The most useful ClineFlow habit is not "write more documentation." It is "leave the next task a truthful starting point." A journal is the working record for one meaningful slice of work. It can be

[12] ClineFlow repository and its documented agent-agnostic setup workflow.

an engineering task, a writing revision, a design investigation, or a research question. It should be short enough to update while the work is fresh and structured enough that a new reader can find the important parts.

You do not need to know or manage ClineFlow's project rules, context maps, decisions, source records, or file paths. Installation already teaches the agent how to use them. Speak normally about what you want. The one continuity concept worth naming is the **journal**: say "start a journal for this task" when work is genuinely new, "check the journal" when you return, and "update the journal, then please commit" after a verified result. ClineFlow handles the rest in the background. Your participation is intentionally small: state the outcome, approve meaningful scope, inspect the result, and correct anything the agent misunderstood.[13]

That order is more valuable than an exhaustive transcript. A transcript says everything that was said. A journal says what the project should remember.

For example, a portfolio website journal might state:

```
Goal: Make the case-study cards readable on a phone.
Confirmed: The cards are rendered by PortfolioGrid in src/components.
Decision: Keep each card's title, role, outcome, and link visible without hover.
Verification: Tested at a narrow browser width with keyboard navigation.
Open: Image descriptions still need author review.
Next: Check the homepage featured-project order.
```

An agent that reads this note can locate the right component, preserve the rule against hover-only information, and avoid claiming that the image descriptions are done. A human reader can challenge the result without searching an entire chat archive.

[13] ClineFlow repository and its documented agent-agnostic setup workflow.

The automation contract: your goal in, durable context out

ClineFlow is designed to remove documentation ceremony, not add it. The installer creates the agent-facing configuration and the OKF knowledge bundle. The agent reads those instructions at the start of work, creates a task journal when one is needed, writes down decisions and verification as the task evolves, and pairs the context update with the implementation at commit time. The repository describes this as automatic journaling and self-documenting work, not a separate note-taking workflow.[14]

That does **not** mean you surrender responsibility to automation. The human still supplies the purpose, decides whether a proposed action is in scope, reviews changes, and owns high-stakes decisions. ClineFlow automates the otherwise tedious mechanics: creating the right project-memory files, finding the active context, preserving the agent's discoveries, and making the next session start informed.

For most tasks, the interaction can be this short:

```
Build [desired outcome] in this project. If this is new work, start a journal;
otherwise check the journal and continue. Propose a plan before changing
anything. Keep project memory updated automatically. Ask me only when you need
a decision, approval, or missing source material.
```

Then work normally. Say what you want to build, answer a real question when it matters, review the proposed change, and say "please commit" when the result is ready. The agent handles the context plumbing in the background. If you open the `knowledge/` folder, you can inspect its work; you should not need to author its files just to benefit from durable memory.

This is why the same workflow works for a developer, a writer, and a lawyer. The developer does not hand-maintain a changelog; the writer does not hand-code a story graph; the lawyer does not

[14] ClineFlow repository and its documented agent-agnostic setup workflow.

hand-format a case journal. Each gives the agent an approved objective and reviews the evidence appropriate to the work. ClineFlow makes the trail durable and discoverable across the next session.

In practical terms, expect the automation to do five things after setup:

1. Generate the agent instruction files and OKF entry points for the project.
2. Find the relevant active journal—or create one when the task is genuinely new.
3. Capture decisions, discoveries, verification, and the next safe step while the agent works.
4. Run the workflow's structural checks when requested and explain any failure.
5. Pair the changed work with its updated context when you ask the agent to commit.

The reader never needs to choose a filename for a decision record before an idea can move forward. If the agent needs information it cannot infer safely, it asks a question. If it knows enough to proceed, ClineFlow handles the context artifacts in the background. That is the promised ease of use: your conversation stays focused on the outcome while the project's memory stays up to date.

The result is a transparent, local, database-like memory layer expressed in readable OKF files—not a hidden vendor database and not another manual workflow. You can open the project and inspect what happened, but you do not have to maintain it. Finish a verified slice by saying **"please commit."** ClineFlow pairs the change with its journal and handoff, so you can close this chat, open a fresh one, switch from Cline to Codex or another compatible agent, or move between local and cloud models without starting the project story over.

Automation should also be visible rather than mysterious. At the end of a task, ask the agent for a one-paragraph handoff: what it changed, what it verified, what remains uncertain, and what it saved for the next session. You are reviewing a concise report, not reconstructing the workflow or writing a second version of

the same documentation. If the report is wrong, correct the agent once; ClineFlow preserves that correction in the project memory for the next task. The benefit compounds precisely because the reader's input is about meaning and approval, while the system owns the repetitive filing.

The simple rule for the rest of this book is: **you ask for outcomes; ClineFlow maintains the context.** Whenever an exercise mentions a journal, decision, or knowledge record, treat it as something the installed agent creates and updates as part of the task. You inspect it when it matters; you do not have to build it from a blank file first.

Validate what the project remembers

ClineFlow documents two local checks with complementary jobs. `validate-okf` checks the knowledge bundle's basic structural expectations. `clineflow-doctor` checks that the project has the required workflow pieces, including the shared instructions and knowledge entry points. The repository describes a default Bash-only path and an optional stricter YAML parsing mode when the extra parser is available.[15]

Run a check after a material setup change and before you hand off substantial work. The goal is not to collect green checkmarks. The goal is to learn whether the next agent will discover the context you intended it to discover.

```
./validate-okf
./clineflow-doctor
```

If a check fails, resist the urge to ask the agent to "fix everything." Read the message. Is the index missing? Is the journal malformed? Did a team rule move? Make the smallest change that restores a clear workflow, then re-run the relevant check. This keeps the validation record understandable.

[15] ClineFlow repository and its documented agent-agnostic setup workflow.

Git makes the memory reviewable

When a task produces both a feature and a journal update, review them together. The code tells you what changed. The journal tells you why, how it was checked, and what remains. A Git commit becomes a useful handoff point because it includes both the product work and the project memory.

For non-code work, the same principle applies. A writer can commit the revised chapter and its story-bible decision. A legal professional can keep a fictional training exercise and its source map together. A designer can commit an accessible component change with the verification note. Git does not confer truth; it preserves the ability to inspect the trail.

Common setup failures

- **Running installation in the wrong folder.** Confirm the project path before installing or letting an agent run commands.
- **Bypassing the installed workflow.** Keep working in the project folder so ClineFlow can use its installed rules.
- **Treating a journal as manual filing.** Ask ClineFlow to start or update it for you.
- **Recording conclusions without checks.** State what was verified and what was merely proposed.
- **Treating a doctor command as a quality guarantee.** It validates setup, not product correctness.

Build checkpoint: the first real task

Choose a task small enough to review in one session. Ask the agent to locate and explain the relevant files, agree on success, and authorize one change. ClineFlow updates the journal and verification trail automatically; then say “please commit” if the result is ready. You have used ClineFlow as intended: a low-cere-

mony memory layer that makes context survive without turning you into a documentation operator.

One workflow, several agents

The agent interface may change, but the operating pattern should remain stable. A Cline, Codex, Cursor, Copilot, or Windsurf user can ask the same ordinary question—"check the journal and continue"—because ClineFlow handles project rules and durable context behind the scenes. The point is not to force all tools to look the same. The point is to ensure that the project's memory is not trapped in one tool's private history.[16]

Use a short opening ritual with any agent:

1. Open the project folder, not a loose individual file.
2. Say "check the journal and help me continue."
3. If this is genuinely new work, say "start a journal for this task."
4. Ask the agent to summarize the relevant context and uncertainty.
5. Agree on the next safe step before authorizing a change.

This sequence may feel slower than typing "build it." In practice, it saves time whenever the system is larger than a single file or the consequences of a wrong assumption are expensive. The agent gets a map. You get a chance to correct the map. Both sides work from a more accurate model of the project.

The grounded prompting loop

The following prompt is intentionally reusable. Adapt the nouns, but keep the order:

```
1. Locate the relevant files, call sites, or source records.
2. Explain how the current system works, using a simple diagram if helpful.
3. Identify assumptions, confusing areas, and any missing context.
4. Propose a plan; do not implement it yet.
5. Define observable success criteria and constraints with me.
```

[16] ClineFlow repository and its documented agent-agnostic setup workflow.

```
6. Ground each planned change in the actual project.
7. Implement the agreed, smallest useful slice.
8. Verify the result and report the evidence.
9. Have ClineFlow update the active journal with decisions, checks, and the next
step automatically; show me the handoff.
```

The order matters. A plan built before locating the real source of truth is often a polished version of the wrong idea. Asking for an explanation gives you a chance to notice that the feature is controlled by a different component, an older decision is no longer current, or a supposedly small change has hidden dependencies. The prompting article behind this workflow calls this building a common language before action: you and the agent should be talking about the same files, pathways, and measures of success.[17]

For a non-developer, "call sites" can mean the documents, templates, or instructions that actually control a result. A writer might locate the scenes where a character's history appears. A lawyer might locate the fictional source documents, issue list, and governing template. A designer might locate the component, content source, and accessibility rules. The structure of the question stays the same: what is real now, what are we changing, and how will we know whether it worked?

Avoid over-delegation

An agent can propose commands, edit many files, and make confident claims. That does not mean it should decide project scope, accept a legal conclusion, publish a site, or handle private material without human review. Keep authority close to the person responsible for the outcome. Use the agent to map, draft, test, and document. Use human judgment to choose the goal, approve tradeoffs, and decide when a result is ready.

This boundary is what makes ClineFlow useful beyond programming. The knowledge bundle keeps a visible record of what the human decided and what the agent generated. That record enables review; it is not a substitute for it.

[17] Hassan Uriostegui, "Stop Telling AI Agents What to Do."

Chapter 2 practice

Create a new folder called `context-practice` and open it with your agent. Tell it, “Please follow the installation guide at https://github.com/hassanvfx/clineflow and install ClineFlow.” Then ask it to initialize Git if appropriate and bootstrap the smallest knowledge bundle with an active journal and project brief. Do not write those files yourself and do not ask it to build a product yet.

Then give the agent one success criterion that a skeptical reader could check. For example: “The active journal names an open question, and a new agent can state the next safe step without inventing history.” ClineFlow runs its structural check and records the outcome; you review it and stop. You have practiced the foundational operation without hand-authoring the context files.

Next step

Chapter 3 adds a local option. You will install and run Ollama on a supported Apple Silicon Mac, measure the model and context it actually loads, and connect that local runtime to an agent. The local model is optional. The durable context workflow is not.

From setup to daily operation

After the first successful check, the workflow becomes pleasantly ordinary. Begin a task by opening the project and asking the agent for its understanding. ClineFlow keeps the active journal current while you work. When a choice matters—an API shape, a data boundary, a story fact, a screen behavior—state it in the conversation and let the agent save it before it gets buried. When you verify something, review the plain-language result it records. When you stop, approve the next safe step it leaves.

This routine answers a common concern: “Won’t documentation slow me down?” It can, if it is written as a retrospective report after the work is forgotten. But a small journal entry made at the moment of a decision is usually faster than reconstructing the decision a week later. It also improves the agent’s next prompt. You no longer need to re-explain the invisible constraints that made the last change acceptable.

Use the following operating card as a chapter-end reference:

```
START
  Say \u201ccheck the journal and help me continue.\u201d
  For genuinely new work, say \u201cstart a journal for this task.\u201d
  Ask the agent to explain the relevant current state.

PLAN
  State the desired outcome and non-negotiable constraints.
  Agree on observable success criteria.
  Ground the plan in real files, records, or call sites.

ACT
  Make the smallest reviewable change.
  Run the narrowest useful verification.
  Record the result, limitations, and next safe step.

HAND OFF
  Review the change and journal together.
  Create a Git checkpoint when appropriate.
```

When to create a new journal

When the question changes enough that a future reader would need a different map, tell the agent the new outcome and let ClineFlow start the new journal. A new feature, a security investigation, a separate chapter revision, or a new fictional matter all deserve their own task record. Continue an existing journal when you are still pursuing the same outcome and the earlier decisions remain central.

Do not create a new journal for every sentence an agent writes. Fragmented context is just another kind of lost context. Prefer one active task record with clear dated updates, then split it only when the subject, authority, or risk changes.

A practical review conversation

At the end of a session, ask the agent four questions:

1. What did you change or learn?
2. Which claims are verified, and by what evidence?
3. Which assumptions remain?
4. What should the next task do first?

Read the answers against the journal. If the agent claims a test passed, make sure the journal identifies the actual command or visible outcome. If it says a decision was made, make sure the decision reflects your intent. If it cannot identify a next step, that may reveal that the task is actually incomplete.

This review is also a defense against smooth but shallow output. Agents are good at producing plausible summaries. A durable workflow gives you a place to ask, “Where is the proof?” The answer may be a test result, a source link, an inspection note, or an honest statement that the result still needs review.

Your first ClineFlow habit

For the next week, do only one thing consistently: before ending an agent session, ask ClineFlow to leave a next safe step in the active journal. It can be tiny: “Compare the empty-state message at mobile width,” “ask the client to confirm the fictional timeline,” or “check whether the local model stayed on the GPU.” This one sentence turns an abrupt ending into a clean restart.

By the time you reach the project chapters, the pattern will feel natural. The finance dashboard, portfolio, todo apps, fiction project, and legal training assistant each use different tools, but they all benefit from the same discipline: preserve the context that the next human or agent must not have to rediscover.

Troubleshooting without losing the plot

When the setup does not behave as expected, return to the smallest observable fact. Do not begin by reinstalling everything. Ask: Which folder am I in? Is this the project folder? Is there a current journal, or is this genuinely new work? What is the smallest safe recovery step?

These questions sound basic because they are basic. They solve a surprising number of problems. Agentic tools often fail in ways that appear mysterious when the project boundary is unclear. A missing file, a stale branch, a different working folder, or an ignored instruction can create behavior that looks like an AI failure when it is really a context failure.

Use this recovery prompt:

```
Do not make changes yet. Check the journal and inspect the current project.
Explain the relevant task, what is missing or inconsistent, and the smallest
safe recovery step. If this is genuinely new work, start a journal first.
```

If the agent cannot answer from the files, that is evidence—not a reason to invent context. Ask ClineFlow to create a short recovery journal, save what is known, and keep uncertainty visible. For example, “The project has an older `docs/journals/` folder and a newer `knowledge/` folder; new work will use `knowledge/`, while the older folder is historical context.” The ClineFlow repository explicitly preserves legacy journals as read-only history while directing new work to the OKF-oriented knowledge bundle.[18]

The difference between a check and a decision

It is easy to confuse a technical check with an approval. `validate-okf` can tell you whether the knowledge structure has the expected shape. `clineflow-doctor` can tell you whether expected workflow files are present. Neither command can tell you whether a financial calculation is fair, whether a portfolio accurately represents

[18] ClineFlow repository and its documented agent-agnostic setup workflow.

a person, whether a fictional scene works, or whether a legal summary is safe to rely upon.

Keep those categories explicit:

Question	Appropriate evidence
Can the next agent find the project knowledge?	Index and structural validation
Did the code behave as intended?	Test, build, or visible inspection
Is this product or creative decision right?	Human review and stated criteria
Is a legal conclusion sound?	Qualified legal review, never an agent assertion

The table is not bureaucracy. It prevents a familiar failure mode: a passing command is described as if it settled a human judgment. A good journal records both kinds of information without confusing them.

End-of-chapter exercise

Take a small project you already have. Ask ClineFlow to create a five-line journal entry using the headings Goal, Confirmed, Decision, Verification, and Next Step. Ask your agent to read the generated entry and summarize what it must preserve. Then ask a second question: “What do you still need to know before acting?”

If the agent answers with a short, accurate map and a reasonable unknown, the workflow is functioning. If it answers with broad confidence but cannot point to the relevant file or source, improve the journal and index. Repeat until the project has a starting point that makes sense to you. That is the success condition for this chapter.

Keep the first version modest

Your first ClineFlow setup does not need a perfect taxonomy, a complex automation, or a giant archive of past work. A readable index, one active journal, a clear project rule file, and one honest

verification note are enough to prove the pattern. Add more structure only when it reduces a real source of repeated explanation.

That restraint is important for teams as well as individuals. If every new task begins with a complicated ceremony, people will work around it. If the workflow helps them answer “what did we decide, what did we verify, and what happens next?” they will keep using it. Durable context compounds only when it remains easy to maintain.

A beginner glossary for the workflow

Several words recur in this book. Use this page as a reference rather than trying to memorize everything at once.

Repository: a project folder whose history is tracked by Git. It can contain code, writing, images, settings, and the knowledge bundle.

Commit: a named checkpoint in that history. It captures a set of changes that can be reviewed or compared later. A commit is not an approval stamp; it is a durable record.

Markdown: a lightweight text format that uses headings, lists, links, and code blocks. It is readable as plain text and renders well in many editors and on GitHub.

YAML frontmatter: a small block of structured fields at the top of a Markdown document. In an OKF concept, it can identify the document type, title, sources, status, or tags.

Journal: a task record that preserves the goal, what was discovered, decisions, verification, open questions, and the next safe step.

Validation: a repeatable check of a particular condition. A structure validator may confirm that expected files or fields exist; a test may confirm a behavior. Neither replaces human review of whether the right thing was built.

Agent: a tool that can reason over project material and, with appropriate approval, propose or perform actions. The agent is a collaborator in the workflow, not the owner of the project's authority.

Context: the information needed to make a good decision. It includes current files, past decisions, sources, constraints, and tests—not only the text inside an active conversation.

A safe permission habit

Before authorizing an agent to run a command or edit multiple files, ask it to state the proposed action in one sentence. For example: "I will add a journal entry, update the portfolio card component, and run the unit test for that component." If that sentence surprises you, stop and clarify the scope.

This habit is especially helpful for beginners because it turns a large technical action into a visible choice. It is also helpful for experienced developers because it catches scope creep before it becomes a difficult review. ClineFlow's journal gives that proposed action a place to live beside the result.

The chapter's promise

You do not need to turn every project into a documentation project. You need enough durable context to prevent repeated misunderstanding. Install the workflow, open with discovery, make work reviewable, and leave an honest next step. Everything else in this book builds on that foundation.

Before you proceed

Pause and confirm that you can answer these four questions about one small project:

1. Where is the project folder?
2. What instructions and knowledge files should an agent read first?
3. Which task is active, and what is its next safe step?

4. What check would distinguish a working result from a plausible-looking one?

If you can answer them, you are ready for the local-AI chapter. If not, do not add more tools yet. Ask ClineFlow to improve the index or journal until the answers are visible. A fast model connected to an unclear project is still working from unclear context.

One final reminder: do not confuse persistence with permanence. Good project memory changes when evidence changes. A journal can say that a decision was revised, a source became stale, or a test exposed a limitation. The value of Git-native context is not that it freezes the project. It makes the project's evolution inspectable, so a future agent can understand not only the current answer but why an earlier answer changed.

That is the discipline you will carry into every local and cloud-assisted build that follows.

Keep it visible, humble, and current.

Next step

Chapter 3 pairs this durable layer with a local Ollama setup without making local AI a requirement.

Continue the chapter online

Use instructions, decisions, and a journal to plan one bounded task.

Next: Ask ClineFlow to check the journal and propose a plan.

Open the companion lab →

CHAPTER 03

INFINITE CONTEXT AND LOCAL AI ON MAC

Measure the model. Keep the memory.

Reader outcome

Set up and measure a local Ollama workflow on an Apple Silicon Mac, while understanding why portable context also helps cloud-agent work.

Planned sections

- The difference between model context and project memory.
- Ollama installation, `pull`, `run`, `ps`, and context tradeoffs.
- Cline at `http://localhost:11434`.
- A measured 24GB M4/M4 Pro lab and troubleshooting checklist.

> **ClineFlow automation rule.** Once installed, ClineFlow's workflow directs the agent to inspect the project, maintain its OKF context, journal decisions and checks, and prepare the next handoff. You supply the goal, machine limits, approval, and review—not the documentation files.

The same local OKF memory works when you close VS Code, open a new chat, or switch between Ollama and a cloud provider. It is a transparent, Git-native, database-like record of the project rather than provider-specific chat memory. After a verified task, say **"please commit"** and ClineFlow carries the implementation, journal update, and next handoff forward automatically.

Two kinds of context

It is tempting to treat a local model with a large context setting as the answer to every memory problem. It is not. A model context window is the amount of material the model can consider during a particular run. It consumes memory, has practical limits, and

disappears from the model's active attention when the conversation ends. Durable project memory is different: it is the files, journals, decisions, sources, and checks that remain available for the next run.

These two kinds of context work well together. A larger local context can help an agent inspect a focused set of files without constant summarization. A ClineFlow knowledge bundle helps the agent decide which files matter and preserves the conclusions after the run. If the local model must be restarted, or if you switch to a cloud model tomorrow, the durable context is still there.[19]

This distinction keeps the phrase infinite AI context honest. The project can accumulate useful, inspectable memory over time. The model still has a finite working window. Treat the model window as a workbench and the knowledge bundle as the organized workshop around it.

What you need on a Mac

The official Ollama documentation lists macOS Sonoma 14 or newer and Apple M-series hardware as supported for Apple Silicon use. Install the Ollama application using the current official download path. The application exposes the `ollama` command-line tool and stores models locally; models can consume tens or hundreds of gigabytes of disk space, so check storage before experimenting with several variants.[20]

For this chapter's lab, use a Mac with an M4 or M4 Pro and 24GB unified memory. That is enough to learn the workflow with smaller or quantized models, but it is not a promise that every coding model, giant context, editor, browser, and tool server can run together comfortably. Unified memory is shared by macOS,

[19] Official Ollama macOS, context-length, and CLI documentation; Cline local-model documentation.

[20] Official Ollama macOS, context-length, and CLI documentation; Cline local-model documentation.

the model, your editor, and the rest of your applications. The safe habit is to measure rather than assume.[21]

Beginner prerequisites

- A supported Apple Silicon Mac and current macOS.
- Enough free disk space for the model you choose.
- A terminal and a project folder under version control.
- ClineFlow installed in that project so it can maintain project memory automatically.
- A compatible agent such as Cline, configured only after the local runtime works.

Do not begin by connecting an agent to an untested server. First prove that Ollama can run one model locally. Then verify that the agent can see the runtime. Then give the agent a small, well-scoped task. This staged approach makes it much easier to tell whether a failure is caused by installation, memory pressure, provider configuration, or the project itself.

Install, pull, and run

After installing Ollama, open a new terminal and use the current model catalog to choose a modest model appropriate for your machine. Ollama documents the basic command pattern:

```
ollama pull <model-name>
ollama run <model-name>
```

The first command downloads a model. The second starts an interactive session. Type one short prompt, confirm that you receive a response, then exit. At this point you have validated the local runtime without involving an editor extension, ClineFlow, or a complex project.

List downloaded models with `ollama ls`. Start the service explicitly with `ollama serve` only when it is not already running through

[21] Author-supplied Local Qwen Setup Guide for a 24GB M4 Pro Mac.

the application. The exact model catalog changes frequently, so this book intentionally does not freeze a recommendation into a permanent command. The important choice is the category: start small, prefer a model documented to work with your available memory, and confirm actual behavior before increasing the load.[22]

Context length is a memory budget

Context length is the maximum number of tokens a running model can consider in memory. It is not a measure of intelligence and it is not free. Increasing it allocates more memory, which can make a previously comfortable model slow down, move work from the GPU to the CPU, or compete with your editor and other applications. Ollama's current documentation notes that large-context agent and coding tasks benefit from substantial context, while also warning that larger settings increase memory requirements.[23]

On a 24GB unified-memory Mac, begin with the default context selected by your current Ollama version and model. Do not set an ambitious number just because a model advertises a large theoretical maximum. Start a focused task, observe the actual process, and increase only if the machine remains responsive and the model stays appropriately resident.

The diagnostic command is:

```
ollama ps
```

It reports active models, their size, processor allocation, context, and expected unload time. Look at it while the agent is actually working, not after the model has stopped. The useful questions are simple:

[22] Official Ollama macOS, context-length, and CLI documentation; Cline local-model documentation.

[23] Official Ollama macOS, context-length, and CLI documentation; Cline local-model documentation.

- Is the model loaded as expected?
- How much context did it receive?
- Is it using the GPU as intended, or has it been pushed partly to the CPU?
- Does the computer remain usable with your editor and project tools open?

There is no universal pass/fail number. A 24GB machine may run a modest model with a focused task well while struggling with a much larger model or a much larger context. The author-supplied M4 guide recommends beginning modestly, observing loaded model size and residency, and using the measurement to decide whether a larger context is justified.[24]

The measurement lab

1. Close applications you do not need. Keep your editor and one project open.
2. Run a small local model with a short, concrete prompt.
3. In another terminal, run `ollama ps` while the model is active.
4. Let ClineFlow record the model name, reported size, processor allocation, and context in the active journal.
5. Ask the model to read a limited project slice: one journal and a small set of related files.
6. Record response speed, any errors, and whether the answer correctly identifies the next safe step.
7. Only then experiment with a larger context or another model.

The journal entry matters. It turns “this felt slow” into a repeatable observation. It also prevents you from treating a single good or bad run as a permanent verdict about local AI.

```
# Local model measurement

Model: <record exactly as ollama ps reports it>
Context: <reported value>
Processor: <reported value>
Task: Check the journal and explain the next safe step.
Result: <speed, accuracy, and any limitations>
Decision: <keep, reduce, or retest this configuration>
```

[24] Author-supplied Local Qwen Setup Guide for a 24GB M4 Pro Mac.

This lab reinforces the main argument of the book. The local model carries a finite working set. The journal holds the durable facts that make a small, focused working set useful.

A decision rule that prevents frustration

Do not tune three variables at once. If you change the model, the context length, and the agent task, you cannot tell which change caused the new result. Change one variable, repeat the same small task, and compare the journal records. If performance degrades, return to the last configuration that behaved well.

This method is less exciting than chasing benchmark charts, but it is more useful for a personal workstation. Your aim is not to demonstrate the largest model that can barely start. Your aim is to build a reliable local assistant that can read the right project context, complete bounded work, and leave the machine responsive enough for you to review the result.

Connect Cline after the runtime works

Once `ollama run <model-name>` succeeds in the terminal, configure the agent. Cline's local-model documentation describes choosing **Ollama** as the provider, using the local base URL `http://localhost:11434`, and selecting an installed model. It also recommends compact prompts and focused tasks for local inference.[25]

The order is worth preserving:

1. Verify Ollama locally in the terminal.
2. Open Cline settings and choose the Ollama provider.
3. Confirm the base URL is the local address.
4. Select the model you already pulled.
5. Ask Cline a small read-only question about the current project.
6. Inspect `ollama ps` while it answers.

[25] Official Ollama macOS, context-length, and CLI documentation; Cline local-model documentation.

If the connection fails, do not immediately change model settings. Check that Ollama is running, that the base URL is local and correctly typed, and that the model is present in `ollama ls`. Those basic checks isolate the connection issue from the model-quality question.[26]

A grounded local-agent prompt

Use the local model for an intentionally small first task:

```
Check the journal. Do not edit files. Explain the current goal, the confirmed
facts,
the open question, and the next safe step in no more than six bullets.
```

This prompt gives the model a narrow context target. It does not ask it to ingest an entire repository or invent a solution under memory pressure. Compare its answer with the journal. Did it name the right goal? Did it distinguish a verified fact from an assumption? Did it preserve the next safe step? If not, improve the index or reduce the requested scope before blaming the model.

When the answer is accurate, proceed to a small planned change. Ask the agent first to identify the relevant files and describe the proposed edit. State the success criteria. Let it make one slice of work. Then run a relevant check; ClineFlow updates the journal automatically. This is the same ClineFlow workflow from Chapter 2; the local runtime changes where inference happens, not how you preserve project judgment.

Run Ollama inside VS Code with Cline

The terminal test proves that the runtime works. The next goal is a full IDE workflow in which Cline can read the project, propose a plan, make a controlled change, run a check, and leave a durable journal entry. In VS Code, install the official Cline extension from the Extensions view, open the project folder—not just an individ-

[26] Official Ollama macOS, context-length, and CLI documentation; Cline local-model documentation.

ual file—and open the Cline panel. In Cline's provider settings, choose **Ollama**, use the local base URL `http://localhost:11434`, and select a model that you have already verified with `ollama run`. Cline's current local model guide also calls out **Use Compact Prompt** as an important local-model setting.[27]

Do not treat that last setting as a quality switch. Compact prompts reduce the agent's routine prompt overhead so a local model has more room for the project slice that matters. It does not turn a weak model into a strong one, and it does not replace a journal. ClineFlow finds the right context automatically; the journal preserves the conclusion after the task ends.

For the 24GB Mac in this chapter, start with one local model for both planning and implementation. Cline's current local reference configuration begins at 32GB RAM for a 4-bit 30B coding model. That is useful context, not a target to force onto a 24GB machine. A smaller or more aggressively quantized model with a focused task is the safer first experiment. Keep a cloud model available for broader architecture work when the local baseline becomes slow or unreliable. Measure the actual allocation with `ollama ps`; do not infer it from a model name or a download size.[28]

Plan first, then Act

Cline's **Plan** mode is the right place to establish shared understanding. It can inspect and search the codebase and discuss an approach, but it cannot edit files or run commands. In **Act** mode, Cline keeps the planning conversation and can propose edits, run commands, and execute the agreed work, subject to the approval behavior you configure. For a feature that touches more than one file, use Plan before Act; return to Plan if the work exposes an assumption that the plan did not cover.[29]

This maps directly to the ClineFlow loop:

[27] Official Cline documentation for local models, Plan & Act, IDE use, Auto Compact, `.clineignore`, and MCP, refreshed for this edition.

[28] Official Cline documentation for local models, Plan & Act, IDE use, Auto Compact, `.clineignore`, and MCP, refreshed for this edition.

[29] Official Cline documentation for local models, Plan & Act, IDE use, Auto Compact, `.clineignore`, and MCP, refreshed for this edition.

1. In **Plan**, say “check the journal and help me understand this task.” Ask Cline to name the controlling work, open questions, risks, and a small success criterion. Do not ask for edits yet.
2. Compare the plan to the journal. Correct any missing constraint or stale assumption. Ask for a short ordered implementation plan and the verification command or observable check for each step.
3. Switch to **Act** only when the plan is bounded. Approve one change slice at a time, inspect the diff and result, then let ClineFlow update the journal with what was actually verified.

Use this first Plan prompt in the VS Code panel:

```
Check the journal and help me understand this task. Do not edit files or run
commands. Locate the smallest relevant implementation slice for this task.
List confirmed facts, open questions, affected files, risks, and one
observable success criterion. Finish with a numbered plan of at most five steps.
```

When the plan is accurate, switch the Cline toggle to **Act** and send a deliberately narrow authorization:

```
Implement only steps 1 and 2 of the approved plan. Before each write,
state the file and intended change. Run only the agreed verification.
Do not widen scope. When finished, summarize the result, limitations,
and the exact journal update I should review.
```

The key is that Plan is not a performance ritual. It is the place where a local model’s limited working window is spent on understanding instead of guessing. Act is not permission to abandon review; it is the execution phase of a plan whose boundaries are already visible.

Context and MCP gotchas

Three settings are easy to confuse. First, **context length** is Ollama’s memory allocation for the model. More context consumes more unified memory; use the Ollama app setting or `OLLAMA_CONTEXT_LENGTH` only after a measured baseline shows that the Mac remains responsive. `ollama ps` is the check for the context

allocated to the running model and for CPU/GPU offloading.[30]

Second, **Compact Prompt** and **Auto Compact** are Cline context-management features. Compact Prompt reduces routine prompt overhead for local work. Auto Compact summarizes a long task as it approaches the context limit. Both are useful, but neither is a substitute for project memory: a journal is the human- reviewable source of truth for decisions, verification, and the next step. Keep tasks focused so the summary does not have to carry an entire repository.

Third, **MCP** adds external tools, prompts, and resources to Cline. Cline does not install MCP servers by default. Add only the server needed for the present task, inspect what it can access, and keep credentials and destructive actions behind approval. Each added tool can bring descriptions and results into the working conversation, so an unnecessary MCP server is both a security surface and a context cost. For a first local IDE setup, use no MCP server. Add one read-only, project-relevant integration later only when you can state its purpose, data boundary, and verification method.[31]

Create a `.clineignore` before opening a larger repository. Exclude generated build output, dependency folders, caches, and secrets such as `.env*`; do not exclude the small `knowledge/` records that make the project navigable. Cline uses this file when it scans paths for context. The goal is not to hide the repository. It is to prevent irrelevant files from pushing the active journal, constraints, and source code out of the model's useful working set.[32]

Local and cloud are complementary

You do not have to run a local model to use ClineFlow. A cloud model may be the practical choice when you need stronger rea-

[30] Official Ollama macOS, context-length, and CLI documentation; Cline local-model documentation.

[31] Official Cline documentation for local models, Plan & Act, IDE use, Auto Compact, `.clineignore`, and MCP, refreshed for this edition.

[32] Official Cline documentation for local models, Plan & Act, IDE use, Auto Compact, `.clineignore`, and MCP, refreshed for this edition.

soning, a large temporary context, capabilities your local model lacks, or simply a faster workflow. ClineFlow's value remains the same: the important facts are stored in project files rather than being trapped in one provider's conversation history.

Local inference can be attractive when you want to experiment privately with synthetic material, work without a network connection, or control the runtime on your own machine. But local does not automatically mean private enough for every situation. A project can still contain sensitive source files, an agent can still edit the wrong path, and an output can still be wrong. Apply the same access controls, review steps, and data boundaries you would use with any other tool.

For the legal training exercise in Chapter 9, this distinction is essential. Use fictional material even if the model runs locally. The example teaches organization, provenance, and review; it does not authorize real-client data handling or legal analysis.

Cloud continuity test

After a successful local task, switch temporarily to a cloud agent —or simply imagine a teammate using a different tool. Give it the same natural request: "check the journal and help me continue." If it identifies the current state and proposes the same safe next step, the system has passed an important test: the project memory is portable.

The model may differ in speed or style. The shared knowledge should not.

That is the durable advantage of the workflow.

When a local setup struggles

Most local-model problems are ordinary systems problems. The model may be too large for the available memory. The context setting may be too ambitious. The service may not be running. The agent may point at the wrong URL. A previously successful

configuration may become slower because the browser, editor, or another application is consuming more memory.

Work through failures in this order:

1. Confirm the Mac is supported and has sufficient free disk space.
2. Run the model directly in the terminal before involving an agent.
3. Use `ollama ls` to verify the model is installed.
4. Use `ollama ps` during an active request to inspect allocation.
5. Reduce the task scope before changing the model.
6. Reduce context or choose a smaller model if the machine is unresponsive.
7. Re-check the agent provider and local base URL.
8. Let ClineFlow record the observation and the next experiment in the journal.

Avoid the common response of downloading several models and changing many settings at once. That consumes disk space and produces a confusing trail. A measured workflow gives you a configuration you can return to.

A local-AI journal entry

```
---
type: Engineering Journal
title: Local model trial for finance dashboard
status: draft
---

# Goal

Use a local model to explain the current dashboard task before editing.

# Configuration

Runtime: Ollama on local Mac
Model: <exact installed model name>
Context: <reported allocation>

# Observation

The model read the active journal and identified the empty-state task.
Response was usable, but the editor became slow with a larger context.

# Decision
```

```
Keep the smaller context for focused journal-and-component tasks.

# Next step

Use a cloud agent for a broader codebase review, while preserving the same
journal.
```

This is not a benchmark. It is a decision record for your actual work. A different Mac, model release, or project can lead to a different result. The journal keeps that distinction visible.

Reader checklist

Before you begin a meaningful local-agent task, confirm:

- Ollama can run the chosen model in the terminal.
- You have checked disk space and current machine responsiveness.
- The project folder is open to the agent and the journal names a bounded goal.
- The active journal defines a bounded goal and next safe step.
- The agent uses the local provider at the expected local URL.
- You know which verification will determine whether the task succeeded.
- Any sensitive or real-world material has been excluded unless you have an appropriate authorized workflow.

If any checkbox is unclear, return to the smallest test. A local runtime becomes useful through repeatable, supervised work—not by being maximally complex.

The most useful stop condition

Stop increasing model size or context when the setup stops helping the work. If an agent can accurately read the active journal, identify the relevant component, and complete a small verified task, you have a usable local workflow. More capacity may be worthwhile later, but it is not a prerequisite for the durable-context practice taught here.

This stop condition protects attention as well as memory. It keeps the project goal at the center: build and review something useful, then preserve what you learned.

Document the configuration that works, including its limits. A reliable modest setup is more valuable than an impressive one you cannot reproduce tomorrow.

A complete first local task

Use this exercise to combine the chapter's ideas without putting a real project at risk. Open a dedicated project folder called `local-context-lab` with the agent and say, "Please follow the installation guide at https://github.com/hassanvfx/clineflow and install ClineFlow." Then ask it to bootstrap a fictional finance-dashboard task. Tell it that the dashboard uses sample data only, empty categories must display safely, and the first task is to identify the chart component. The agent creates the index and journal automatically; you review the stated constraints before authorizing an edit.

Configure your local agent only after the terminal test succeeds. Then use this prompt:

```
Check the journal. Do not edit files. Locate the likely component or placeholder
responsible for the chart.
Explain the current task, list any assumptions, and define a small
verification step that would prove an empty category is handled safely.
```

If there is no actual component yet, the correct answer is not a fabricated file name. The correct answer is that the project has a stated goal but no implementation. That is still a successful context exercise: the agent read the record, separated fact from absence, and proposed the next safe action.

If you do have a small component, ask the agent to identify it before authorizing an edit. Keep the requested change narrow: show a clear zero-spend message when the data array is empty. Ask the agent to state how it will verify the result. After the

change, inspect the output yourself; ClineFlow records the result and next safe step in the journal automatically.

What success looks like

At the end of the exercise, a fresh agent—local or cloud—should be able to answer the following without reading a long chat history:

- What does the project do?
- Which data is safe to use in the lab?
- What behavior needs improvement?
- Which file or component controls it, if one exists?
- What was verified?
- What is the next safe step?

If the answer is visible in the index and journal, you have combined local inference with durable project memory successfully. If the answer only exists in your head or in a discarded chat, improve the record before increasing the model's context window.

Before using local AI with a larger project

As projects grow, avoid opening every file just because the model can accept more context. Start by asking ClineFlow to check the journal and identify the relevant subsystem. It traces the installed project context and loads only what the present decision needs. This approach helps local models, cloud models, and humans: less unrelated material means fewer opportunities to confuse old history with current behavior.

The same principle is why ClineFlow works when you never install Ollama. The system is not a model-size strategy. It is a project-memory strategy. Local AI is one useful execution environment for that strategy.

Chapter review

You should now be able to distinguish a model's temporary context from the project's durable context; install and test Ollama on supported Apple Silicon hardware; connect a local runtime to Cline; use `ollama ps` to observe actual allocation; and keep local experiments focused, synthetic, and journaled. Keep a cloud option available when it better serves the task. The knowledge bundle makes that choice reversible.

Do not treat a single configuration as universal advice. Let ClineFlow record the machine, model, context, task, observation, and decision. Revisit that record when your tools, project, or hardware changes. That is how a local experiment becomes reusable knowledge rather than another forgotten setup conversation.

Measure first, decide second, and preserve the reason.

Keep every experiment small, reviewable, and reversible.

Next step

You now have three layers: a portable OKF-shaped knowledge bundle, ClineFlow's working memory and verification habit, and an optional local runtime. Chapter 4 turns the theory into a concrete build: a private personal-finance dashboard using synthetic data. ClineFlow creates the project map before the interface, you define success before code, and the journal carries context automatically across each iteration.

A reproducible local-runtime baseline

Before you compare models or providers, create one baseline that another future you can repeat. Choose one project slice, one short discovery prompt, one context setting, and one observation

interval. For example, ask the agent to check the journal and report the current goal and next safe step; then use `ollama ps` while it works. Record only what you saw: model name, reported context, processor allocation, task result, and whether the Mac stayed responsive.

This baseline is more useful than a general claim that a model is “fast” or “slow.” A response can feel quick when it is short but still misunderstand the project. Conversely, a slightly slower run that accurately locates the current decision and leaves the machine usable may be the better configuration for a real work session. Separate the task result from the runtime observation in the journal.

```
# Baseline comparison record

Project slice: active journal and the ClineFlow-selected task context
Prompt: Identify goal, confirmed facts, and next safe step without editing.
Model: <exact local model name>
Context reported by ollama ps: <value>
Processor allocation reported by ollama ps: <value>
Result: <what it identified correctly and what it missed>
Responsiveness: <usable / degraded / unusable>
Decision: <retain baseline, reduce scope, or retest one variable>
```

Do not fill a field with an estimate. If you did not observe a value, write “not observed.” That small discipline is the same provenance habit used in the legal and fiction chapters: an unknown remains visible rather than becoming a confident-looking memory. It also means you can compare later runs without mistaking a recollection for evidence.

Change one variable deliberately

When you want to test a larger context, retain the same model and prompt. When you want to test another model, retain the same project slice and record the new model name exactly. When you want to see whether an agent integration adds overhead, first repeat the terminal task, then try the local provider with the same small discovery request. The order matters because it keeps installation, runtime, project context, and agent behavior from becoming one impossible-to- diagnose bundle.

If the result becomes less reliable, return to the baseline. Check whether the agent read the intended files, whether the journal is concise enough to locate, and whether the task loaded unrelated material. More tokens are not always the fix. A clearer index, smaller request, or better decision record may improve the work without consuming additional unified memory.

A privacy and review reminder

Running inference on your Mac changes where the model executes. It does not remove the need to inspect prompts, source files, edits, or outputs. A local agent can still misunderstand a requirement, read a sensitive file available in the project, or make an unsafe change. Use synthetic lab data, narrow paths, read-only discovery prompts, version control, and the same human verification you would use with a cloud agent.

That is the practical meaning of “infinite context” in this book: not an unbounded runtime, but a repeatable ability to recover the current project state from well-kept records. Local and cloud agents both benefit from that discipline.

A small comparison rubric

Use the same four questions after every baseline or experiment: Did the agent identify the current goal? Did it name the controlling files? Did it distinguish confirmed facts from open questions? Did the Mac remain usable while it worked? Write the answers beside the runtime measurements. A configuration that fails the first three questions is not rescued by a larger context number. A configuration that passes them consistently is worth preserving, even if it is not the largest model your hardware can start.

If two configurations both pass, choose according to the work in front of you. The smaller one may be ideal for a focused journal review; a stronger cloud model may be appropriate for a broader planning pass. The durable project record lets you make that trade without losing the thread of the work.

Continue the chapter online

Record a local runtime baseline from observed values.

Next: Ask ClineFlow to check the journal and measure this local run.

Open the companion lab →

PART II

BUILD WITH DURABLE CONTEXT

Turn context into useful work.

CHAPTER 04

PERSONAL-FINANCE DASHBOARD

Build locally. See clearly.

Reader outcome

Build a private, sample-data HTML dashboard while keeping its decisions and checks in a project journal.

Planned sections

- Synthetic data and privacy boundary.
- Prompt sequence and success criteria.
- Dashboard build, calculations, charts, and verification.

> **ClineFlow automation rule.** After installation, ClineFlow's workflow scaffolds and maintains the project memory, journal decisions and checks, and leave a next-session handoff. You give it the outcome, privacy boundary, approval, and review; you do not build the knowledge files yourself.

This makes the dashboard's context a transparent local record rather than a second task for you to administer. You can change chats or agents without re-explaining the privacy boundary: the next agent reads the same OKF memory. When a checked slice is ready, say **"please commit"**; ClineFlow pairs the implementation with its journal, verification, and next safe step.

Build a dashboard without exposing financial data

This chapter creates a small browser-based personal-finance dashboard. It is a learning project, not banking software, investment advice, tax advice, or a secure financial-record system. Use only sample data that you invent. Do not paste account numbers, statements, credentials, or another person's transactions into an agent prompt, a local model, or a companion project.

The finished lab will show a fictional month of income, planned category budgets, and expenses. It will calculate how much was spent in each category, compare that amount with the planned limit, and display an empty state when a category has no transactions. The project will run in a browser from local files. It will not connect to a bank, upload data, or make financial recommendations.

Prerequisites

- A project folder with Git and ClineFlow installed.
- A code editor and a modern web browser.
- A compatible agent, local or cloud.
- Comfort editing small text files; no prior finance or charting experience is required.

Open a folder named `finance-dashboard-lab` and say, “Please follow the installation guide at https://github.com/hassanvfx/clineflow and install ClineFlow.” Then give the agent the privacy boundary below. Do **not** create this map yourself: ClineFlow creates the appropriate project memory and starter files after it reads its generated configuration.

```
finance-dashboard-lab/
  knowledge/
    index.md
    journals/
      dashboard.md
    decisions/
      synthetic-data-only.md
  index.html
  styles.css
  app.js
```

The generated HTML, CSS, and JavaScript files are not a product yet. They are a boundary. They make it clear that the dashboard will be a small local web page, not a request for the agent to create accounts, call third-party APIs, or choose a financial platform.

Write the project brief first

Tell the agent this intent; ClineFlow saves it to the active journal automatically:

```
# Goal

Build a browser-based finance dashboard using fictional sample data.

# Constraints

- No bank connections, uploads, accounts, or credentials.
- Calculations are illustrative, not financial advice.
- Empty categories show a safe, readable zero state.
- The result runs locally in a modern browser.

# Success criteria

- The dashboard shows planned, spent, and remaining amounts per category.
- Totals update from the sample transaction list.
- A zero-transaction category does not break the interface.
- The active journal records the verification and next step.

# Next safe step

Locate the current files and propose a small page structure. Do not build yet.
```

This is the first practical use of the grounded prompting loop. Rather than asking an agent to "build a finance dashboard," you give it a bounded goal and non-negotiable constraints. You make the privacy boundary part of the project context, not a detail that could be forgotten halfway through the build.[33]

The decision record

Tell ClineFlow's agent to create the `synthetic-data-only` decision record and active dashboard journal automatically before implementation. State that the project accepts only data committed to the example. If you later add a form for practice, it may operate only inside the browser and should explain that entries are temporary examples. The generated decision record gives a future agent a clear answer when it proposes a convenient but

[33] Hassan Uriostegui, "Stop Telling AI Agents What to Do."

inappropriate bank integration or cloud spreadsheet connection: that work is outside this lab's scope.

The same record helps when you share the project. A reviewer can see that missing real-data features are intentional, not incomplete implementation.

Keep this boundary in the index as well. The most important constraint should be discoverable before an agent writes a single integration line.

Visible boundaries are easier to preserve.

They also make a safer, clearer demonstration for every reader.

Always.

Locate and understand before building

Open the folder in your editor. Ask the agent to check the journal and explain the project before writing the dashboard.

```
Check the journal and help me plan this dashboard. Do not edit yet. Explain the
smallest structure, any missing constraint, and how planned, spent, and
remaining values can be calculated from fictional sample data.
```

For this small project, a good answer will likely assign document structure to `index.html`, layout and visual rules to `styles.css`, and sample data plus calculations to `app.js`. The agent should also notice that the project needs an agreed data shape before it can render anything. If it suggests a framework, a server, a database, or a bank API, point back to the brief: those are outside the intended lab.

Use a simple, inspectable data model

Keep the example data small enough to inspect on one screen:

```
const categories = [
  { id: "home", name: "Home", planned: 1200 },
  { id: "food", name: "Food", planned: 450 },
  { id: "transport", name: "Transport", planned: 160 }
];

const transactions = [
  { categoryId: "home", amount: 1200, label: "Sample rent" },
  { categoryId: "food", amount: 78, label: "Sample groceries" },
  { categoryId: "food", amount: 21, label: "Sample lunch" }
];
```

These numbers are deliberately fictional. They demonstrate relationships, not a recommended budget. The `transport` category has no transactions so that you can verify the empty state. A category has a planned amount; a transaction belongs to one category and has an amount; the dashboard can sum matching transactions.

Define the calculations in words before writing JavaScript:

- **Spent** is the sum of sample transactions with the category's ID.
- **Remaining** is planned minus spent.
- **Total planned** is the sum of all planned values.
- **Total spent** is the sum of all sample transaction amounts.
- **Total remaining** is total planned minus total spent.

Writing the definitions first prevents an agent from quietly choosing a different meaning for "remaining." It also makes review possible for a reader who does not write code.

Agree on a minimal interface

The first screen needs only three areas:

1. A heading that says the dashboard uses fictional sample data.
2. A summary row for total planned, total spent, and total remaining.
3. One card per category, showing planned, spent, and remaining.

Do not add a chart until the cards work. Charts can make an unfinished calculation look impressive. The cards are easier to verify because a reader can compare each number directly with

the sample list. Once the cards are correct, a small bar or progress indicator can help show the relationship visually.

Define success before implementation

Have ClineFlow add these checks to the journal automatically:

```
- Food spending is the sum of the two fictional food transactions.
- Transport displays $0 spent without an error.
- Remaining is calculated from planned minus spent.
- The page labels the data as fictional sample data.
- No network request, login, or upload is introduced.
```

Now the agent can make a plan grounded in the files and definitions you just reviewed. Only after you approve that plan should it implement the first slice.[34]

A short plan review

Before approving implementation, ask the agent to restate the plan in three parts: files it will edit, calculations it will implement, and checks it will run. Review each part against the constraints. The plan should not require a package installation, account, remote service, or data source. It should mention the empty transport category explicitly.

If the plan is longer than the project needs, reduce it. A well-scoped first slice is: render the summary and category cards from the fixed arrays, then open the local HTML file and inspect the zero-transaction card. Save charts, filters, editable forms, and persistence for later iterations. The journal should say why they are postponed rather than leaving a future agent to assume they were forgotten.

Implement one reviewable slice

After you approve the plan, use a prompt that keeps the agent accountable to the project record:

[34] Hassan Uriostegui, “Stop Telling AI Agents What to Do.”

```
Implement only the first dashboard slice described in the active journal.
Use the fixed fictional arrays in app.js. Render the summary and category cards.
Do not add network requests, forms, storage, packages, or third-party services.
After editing, explain the calculation for each displayed total and list the
manual browser checks needed to verify the empty transport category.
Have ClineFlow update the journal automatically with the files changed and the
next safe step.
```

The agent should calculate each category's spent amount by filtering or grouping the transaction list by `categoryId`, then summing the amounts. The remaining value should be derived from the planned amount minus the spent amount—not manually typed into the page. This distinction matters because the visual interface should be a result of the data model, not a second source of financial facts.

Verify in the browser

Open the local `index.html` file in a browser. Inspect the page deliberately:

1. Confirm the page visibly says it uses fictional sample data.
2. Add the two sample food transactions by hand and compare the sum with the food card.
3. Confirm the transport card displays zero spent and remains readable.
4. Add planned, spent, and remaining category values; compare them with the summary row.
5. Narrow the browser window and make sure the cards do not overlap or hide their labels.
6. Open the browser's developer tools only if you see an error; do not treat the absence of a visible error as proof that the numbers are right.

This is manual verification, and that is appropriate for a small browser lab. Record what you actually inspected. "Dashboard works" is not enough. "Food card displayed $99 from the two fictional sample transactions; transport displayed $0; summary totals matched the card values at a narrow viewport" is useful evidence.

Add a small chart only after the cards pass

Once the cards are correct, you can ask the agent for a simple visual comparison. A horizontal bar showing spent relative to planned is enough. Keep the chart derived from the same calculation object used by the cards. Do not create a separate, hard-coded set of chart values.

Use a constrained prompt:

```
The category cards passed the journaled manual checks. Add a simple accessible
visual indicator derived from the existing calculation data. Keep the text
values visible, label the indicator, and do not add a charting library.
Describe how the indicator behaves for a zero-spend category and a category
that exceeds its planned amount.
```

The text values remain important because a bar alone is not accessible or precise. If a category exceeds its planned amount, the dashboard can display a clear warning state such as “$25 over plan.” It should not make a recommendation about what the reader ought to do with money.

Journal handoff

Close the first iteration with a record like this:

```
# Confirmed

The local page renders three fictional categories and totals from fixed arrays.
Transport shows $0 spent without an error.

# Verification

Manual browser check: food total matched the two listed transactions;
summary matched category cards; labels remained visible at narrow width.

# Open

The visual indicator needs a keyboard and contrast review.

# Next step

Ask the agent to locate the indicator styles and propose an accessibility check.
```

The next session now begins with the data boundary, completed work, evidence, and an accessible next question. That is the project-memory payoff.

Common failures

- The agent adds a bank API because it assumes “personal finance” means live data. Re-state the decision record and remove the integration.
- Totals are hard-coded into the HTML. Move the calculation into JavaScript so the cards and summary share one source of truth.
- A zero category disappears. Keep the category card visible; zero is meaningful information.
- A chart replaces text. Preserve readable numeric labels and visible category names.
- The journal says “tested” but names no observation. Ask Cline-Flow to record the exact browser check.

Next step

Chapter 5 applies the same discipline to a public-facing portfolio. The data changes from fictional transactions to your own approved project information, but the workflow stays familiar: map the content, define what must remain true, build a small slice, verify it, and preserve the next step.

Accessibility and continuation exercise

The dashboard is small, but it is still an interface. Before adding polish, check whether a reader can understand it without color, hover effects, or a perfect desktop display. Use this review prompt:

```
Check the journal and review the current dashboard for accessibility. Do not
change anything yet. Identify accessibility risks in the summary, category
cards, and
visual indicators. Consider keyboard reading order, visible text labels,
contrast, narrow screens, and the zero-transaction state. Propose the smallest
set of improvements and a manual verification checklist.
```

The agent may identify risks such as a color-only “over budget” warning, a progress indicator with no text value, a heading hierarchy that skips levels, or cards that become too narrow on a phone. Ask it to explain each finding in plain language before editing. A good improvement is one that preserves the existing calculation and makes it easier to interpret.

For example, an over-plan category can include the text “$25 over plan” in addition to a visible color change. A zero-spend category can include “No sample transactions this month” rather than an empty blank. A progress bar can use `aria-label` and retain the numeric planned and spent values beside it. These are not compliance guarantees; they are practical review habits.

Continue without losing the model

Imagine you return next week and decide to add a fictional transaction form. Do not start by asking the agent to “add a form.” Read the journal. The original boundary says no persistence, no remote account, and no real data. A proposed local form should therefore add or remove entries only in the current browser session and should describe itself as a simulation.

Ask ClineFlow to create or update the journal automatically before implementation:

```
# New goal

Allow a reader to add a fictional transaction during the current browser
session.

# Preserved constraints

- No storage after refresh.
- No uploads or remote requests.
- No real financial data.
```

```
# Success

A valid fictional entry updates cards and totals; refresh returns to the fixed
samples.
```

This is the “continue” part of the ClineFlow loop. The original project record prevents a later improvement from quietly changing the nature of the lab. It also gives the agent a clear edge case: browser refresh should reset the simulation.

A final verification card

Before you consider the lab ready, ask ClineFlow to add a short verification card to the journal automatically:

Check	Evidence
Synthetic data boundary	No credentials, network calls, uploads, or accounts in the project
Calculations	Card and summary totals match the fixed sample arrays
Empty state	Transport remains visible with $0 spent
Readability	Text labels remain visible at a narrow viewport
Handoff	Journal names the next safe step and any remaining accessibility review

The card is intentionally small. Its job is to make the result reviewable, not to turn a learning project into a financial product certification. Once it exists, a future agent can locate the relevant files, see what was checked, and propose the next improvement from shared context rather than guessing.

Chapter review

You used a real project pattern without real financial data: define boundaries, map the work, make the model explicit, render a minimal interface, verify a meaningful edge case, and preserve the handoff. The next chapter changes the content from fictional numbers to an author-controlled public portfolio, where truthfulness and presentation become the central constraints.

Do not skip the review because the sample is simple. The point of a safe lab is to practice habits you will later apply to work that matters more: keep claims traceable, preserve important constraints, and give every next session a trustworthy starting point.

Small checks create durable confidence over time.

Use them.

A controlled simulation extension

The optional fictional transaction form is a useful test of whether the project can evolve without forgetting its first decisions. It should operate only in the current browser session. The fixed sample arrays remain the baseline; refreshing the page returns the demonstration to its original state. This gives a reader a visible interaction without suggesting that the dashboard stores a financial record or accepts real money data.

Before code, place the extension decision in the journal and ask the agent to inspect the existing calculation path:

```
Check the journal and inspect the current dashboard. Do not edit. Explain the
smallest way to add a fictional transaction for the current browser session
only. State how refresh resets the sample, how invalid amounts
are handled, and which existing calculation checks must still pass. Do not add
localStorage, a network request, accounts, credentials, uploads, packages, or
financial recommendations.
```

The plan should reuse the same calculation object already behind the cards and summary. A new sample transaction belongs to an existing category, has a simple fictional label, and has a positive illustrative amount. The form should reject an empty label, an unknown category, or a non-positive amount with a plain message. It should not attempt to validate a bank statement, infer a spending category, or remember anything after the reader reloads the page.

Define the extension's success

Write success criteria before accepting implementation:

- A reader can select an existing category and enter an illustrative transaction.
- A valid entry updates that category's spent and remaining values and the summary totals.
- Invalid or blank entries do not alter the calculation data.
- The page continues to label all values as fictional sample data.
- Refresh restores the fixed baseline arrays.
- The zero-transaction category still renders until the reader adds a fictional entry to it.
- No persistence, service, account, upload, or external request is introduced.

Then make the bounded implementation request:

```
Implement only the approved current-session fictional transaction form. Reuse
the existing category calculation and rendering path; do not create a separate
set of totals. Validate empty labels, unknown categories, and non-positive
amounts without changing data. Keep the fixed initial samples after refresh.
After editing, report changed files and run the agreed manual checks. Let
ClineFlow update the
journal with the result and the next safe step.
```

Test the behavior in a sequence

Open the local page and use one invented entry, such as "Sample bus ride" for Transport. Confirm that the transport card changes from zero spent, the summary changes by the same fictional amount, and the category's remaining amount is derived rather than typed. Try a blank title and a zero amount; confirm that

the dashboard leaves the displayed totals unchanged. Reload the page and confirm that the entry disappears and the original transport zero state returns.

Record the sequence, not a general impression: “Added fictional transport item of $12; transport spent became $12; total spent increased by $12; blank label and zero amount made no change; reload restored fixed samples.” This statement is a useful regression baseline for a future agent. It names the behavior, inputs, and result without claiming that the page is financial software.

Extension failure modes

- **Accidental persistence.** The agent adds browser storage because it is common in forms. Remove it; session-only behavior is deliberate.
- **Split calculations.** The form updates one card but not the summary because it uses a second total. Route both through the shared calculation path.
- **Unclear reset.** A reader assumes their entry is saved. State visibly that refresh restores fictional samples.
- **False realism.** Labels or UI copy invite account balances, banking credentials, or investment decisions. Keep the wording educational and invented.
- **Lost edge case.** Transport disappears once the form exists. Preserve the initial zero state and re-check it after every rendering change.

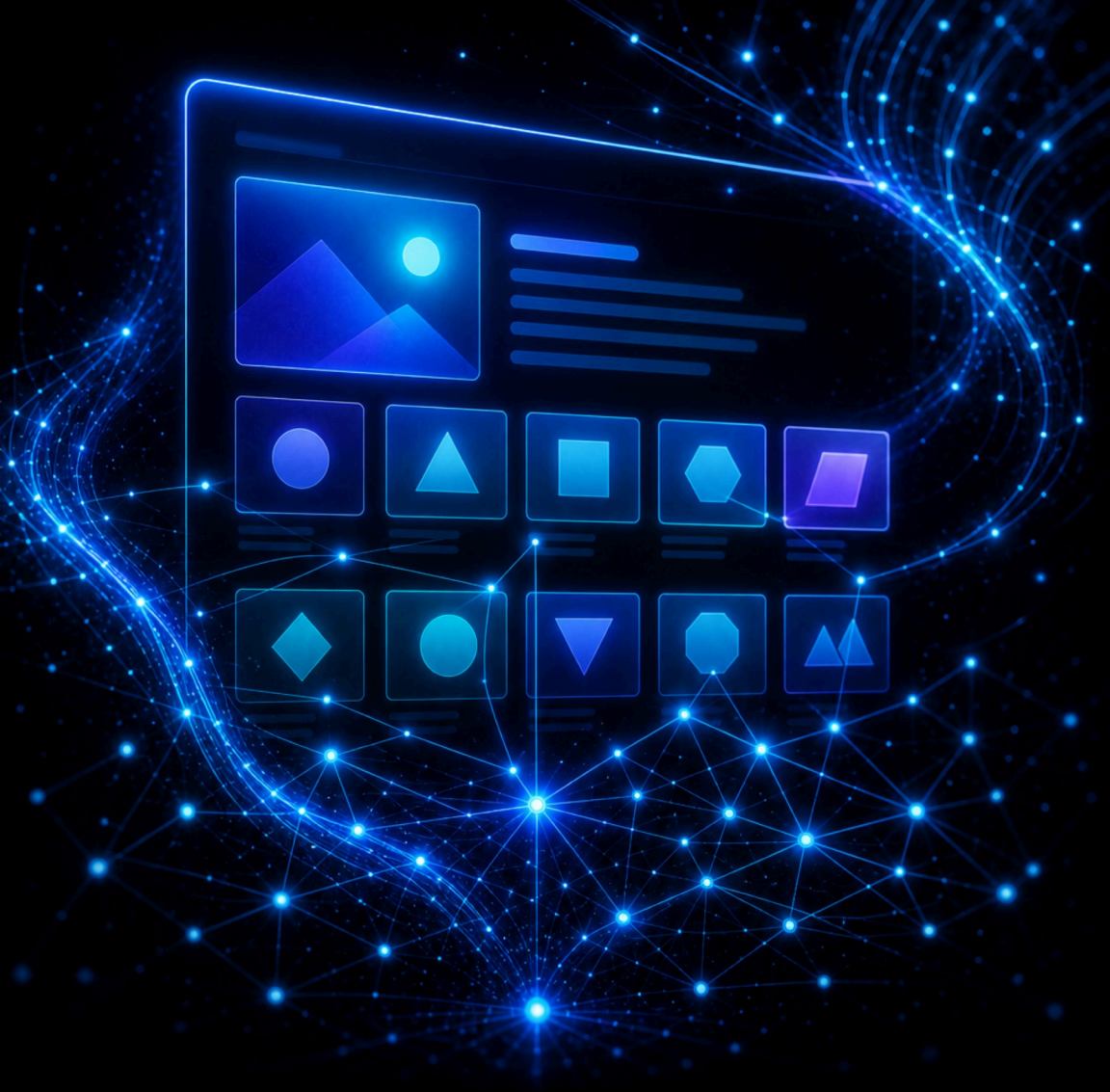
This extension demonstrates the value of a project journal: the new behavior does not erase the original local-only, sample-data, reviewable purpose.

Continue the chapter online

Build a fictional-data dashboard with a visible zero state.

Next: Ask ClineFlow to check the journal and plan the dashboard.

Open the companion lab →



CHAPTER 05

PORTFOLIO WEBSITE

Make public claims you can support.

Reader outcome

Build an accessible portfolio and retain decisions about positioning, content, and revisions.

Planned sections

- Content inventory and voice.
- Responsive structure and accessibility.
- Journal-led review and continuation.

> **ClineFlow automation rule.** Once installed, ClineFlow creates and maintains the content map, decisions, journal, checks, and handoff. You provide approved facts, publication boundaries, approval, and review—not a manual knowledge bundle.

The approved-content record behaves like a transparent, local database for the site: readable OKF files that travel with the project, rather than facts trapped in one chat. You may switch agents without restating every public-claims rule. After you verify a slice, say **“please commit”** and ClineFlow preserves the page change, review record, and next handoff together.

Build from approved facts, not confident guesses

A portfolio is public-facing. That makes context especially important. An agent can create a beautiful page from incomplete information and still introduce a false job title, an exaggerated outcome, a client name you should not publish, or a voice that does not sound like you. The first job is therefore not choosing

colors or components. It is creating an approved content inventory.

Open a folder called `portfolio-lab` and give ClineFlow the approved facts and publication boundary. It creates and maintains a short knowledge bundle such as:

```
portfolio-lab/
  knowledge/
    index.md
    journals/portfolio.md
    references/approved-bio.md
    references/approved-projects.md
    decisions/public-claims.md
  index.html
  styles.css
  app.js
```

The reference files are author-controlled sources. They can contain a short biography, contact method, links the author wants to share, and project descriptions that are true and cleared for publication. If a detail is unknown or confidential, do not add it so the agent can “make the page complete.” Mark it as unavailable.

Reader outcome

You will build a small responsive portfolio that presents approved work clearly, remains usable on a phone, and retains a journal of content decisions and review checks. The project can be static HTML/CSS/JavaScript; it does not require a CMS, account, analytics platform, or publishing service to complete the lab.

Inventory before interface

Give ClineFlow the approved project facts and ask it to create the `approved-projects.md` table automatically:

Project	Role	Approved outcome	Public link	Constraints
Example dashboard	Designer and builder	Local learning demo	none	Use synthetic data only
Example portfolio	Author	Demonstrates responsive layout	none	Do not claim client work

The table is intentionally plain. It makes the content legible to both a human reviewer and an agent. It also separates an approved outcome from a marketing claim. “Demonstrates a responsive layout” is an observable statement. “Transformed the industry” is vague and may be untrue.

Tell the agent this decision; ClineFlow adds it to the journal automatically:

```
# Public-claims rule

The site may state only facts in the approved bio and project references.
Unknown roles, metrics, client names, endorsements, and dates must be omitted
or flagged for author review. The agent must not invent them.
```

The grounded content prompt

```
Check the journal and help me plan this portfolio from the approved project
facts. Do not edit files. Create a content map for a one-page
portfolio: introduction, selected work, contact, and an optional short process
section. Identify missing facts and any claim that needs author approval.
Propose a responsive HTML structure and success criteria; do not write copy
that is not supported by the references.
```

This prompt creates a useful constraint: the agent must make uncertainty visible. A missing project result is not a reason to manufacture a metric. A missing photograph is not a reason to generate a fake portrait. The author remains responsible for the

public message; the agent helps arrange and review the material.[35]

Define success before design

Write success criteria that are visible in the browser:

- The introduction uses only approved bio facts.
- Every project card includes a role and outcome supported by the reference file.
- Contact information is intentional and limited to the author-approved method.
- The site remains readable without hover, at a narrow viewport, and with keyboard navigation.
- No analytics, external tracker, publication action, or invented claim is introduced.

These criteria prevent a common agent failure: treating a portfolio as a generic conversion page. This lab is a truthful, author-controlled presentation. A simple page with accurate claims is more useful than an impressive page that asks readers to trust information the author never approved.

Accuracy is the portfolio's strongest design system.

Keep the record clear, current, and author-approved.

Review before release.

Build the responsive structure

Once the author approves the content map, ask for a small static structure:

```
Implement the approved one-page portfolio structure only. Use semantic HTML:
a header, main content, sections for introduction and selected work, and a
footer or contact section. Render project cards from the approved project data.
Do not add a CMS, analytics, tracking, a contact form, a publication action,
or unapproved copy. Keep all text visible without hover.
```

[35] Hassan Uriostegui, "Stop Telling AI Agents What to Do."

Semantic HTML matters because it gives the page a useful reading order before styling begins. A visitor using a keyboard or screen reader should encounter the author's introduction, selected projects, and contact information in the same logical order as a sighted visitor. Start with headings, paragraphs, links, and lists. Style them after the content is correct.

For a basic card, include the project name, author-approved role, a short supported outcome, and an optional public link. A project with no public link should still be represented honestly. Do not make a fake "view project" button. Absence is better than a decorative action that goes nowhere.

Responsive rules worth writing down

Have ClineFlow add these decisions to the journal before asking for CSS:

- The page has one primary column at a narrow width.
- Project cards may become a grid only when their content remains readable.
- Text does not rely on hover to become visible.
- Links have visible text and a clear focus state.
- Images are optional; if used, their descriptions must be author-approved.
- Motion is optional and should not be required to understand the page.

These choices make the agent's styling work easier to review. Instead of requesting "a modern portfolio," you are defining observable behavior. The agent can propose a grid breakpoint, spacing scale, or color treatment, but it cannot silently trade away a readable mobile layout for a dramatic desktop composition.

Verify the page as a visitor would

Open the local page in a browser and perform a small review:

1. Read every visible claim aloud against the approved source files.
2. Resize to a narrow viewport; check that text, links, and cards remain visible.
3. Use the Tab key to move through links. Focus should remain visible and the order should make sense.
4. Turn off images mentally: does the page still say what the author does and what each project demonstrates?
5. Check external links. Do they point only to author-approved destinations?
6. Confirm that there is no hidden tracker, generated testimonial, or unapproved contact method.

Have ClineFlow record the results in the journal. A useful note is specific: "All project roles matched `approved-projects.md`; the contact link opened the approved site; cards collapsed to one column at narrow width; keyboard focus remained visible." A vague note such as "responsive test passed" forces the next reviewer to repeat the investigation.

When the agent proposes copy

Agents are often helpful at tightening language, but public copy needs an approval loop. Ask for two or three candidate versions and mark them as drafts. Compare each with your source material. Keep a sentence only if it is both true and in your voice. The journal can record a small decision such as: "Use first person in the introduction; avoid unsupported outcome metrics; describe the finance dashboard as a fictional learning project."

This protects the author without making the agent useless. The agent can reveal awkward repetition, propose a clearer hierarchy, and identify missing context. The author decides what becomes a public statement.

Handoff

End the first portfolio session with the same ClineFlow pattern:

```
# Confirmed

The static page renders approved introduction and project cards.

# Verification

Claims matched the approved reference files; keyboard focus and narrow layout
were manually checked.

# Open

Author must review any optional image descriptions and final public wording.

# Next step

Locate the card styles and propose the smallest contrast or spacing improvement.
```

The page is now more than a design. It is an author-controlled public artifact with a traceable content record.

Continue the site without drifting

The first version of a portfolio is rarely the last. You may later add a new project, revise the introduction, change the order of work, or include an author-approved image. Each change should begin with the same question: which approved source controls this statement?

Suppose you want to add a case study. Start by creating an approved project reference, not by asking the agent for a new card. The reference can include the project's public name, your role, a short factual outcome, an optional public link, and explicit exclusions. For example, it may say "Do not identify the client," "Do not publish internal metrics," or "Use only this author-provided screenshot." The agent can then propose a card and a page placement that respects those constraints.

Use this continuation prompt:

```
Check the journal and help me add this approved case study without changing
unrelated claims or layout behavior. Do not edit yet. Explain what facts are
new, what existing language may need revision, and which visual assets are
```

```
approved. Propose the smallest change that adds the project without changing
unrelated public claims or layout behavior.
```

The phrase “without changing unrelated behavior” matters. A small content addition should not lead an agent to rewrite navigation, alter contact information, or replace the page’s visual style unless the author asks for that work. The journal gives you a place to state the intended boundary.

Common failures

- **Invented proof.** The agent writes a metric, customer quote, award, or client name not found in an approved reference. Remove it and add a public-claims reminder to the relevant journal.
- **Design outruns content.** The page contains large empty visual panels because the agent assumed photographs or logos would exist. Prefer a clean text-first layout until an asset is approved.
- **Hover-only meaning.** A card hides its role or outcome until a pointer hovers over it. Keep core content visible.
- **A generic contact form.** The agent adds form handling, email collection, or a third-party service without authorization. A simple approved link is sufficient for the lab.
- **Unreviewed “improvement.”** The agent rewrites the author’s voice while changing layout. Keep copy changes separate from layout changes so each can be reviewed.

A line-editing ritual

Before publication, read the page as the person described by it. Ask: Would I stand behind every sentence? Is the voice mine? Does the project ordering represent what I want readers to see first? Is any information too private, too vague, or too permanent for a draft site?

Then read it as a visitor with no prior context. Can the visitor identify the author, understand the type of work shown, and find an approved way to learn more? If not, make the smallest content

change that fixes the problem. Do not solve a clarity problem by adding unsupported superlatives.

Portfolio verification card

Check	Evidence
Claim provenance	Every visible public statement appears in an approved reference or author review note
Responsive layout	Introduction, project cards, and contact method remain readable at narrow width
Keyboard access	Links have visible focus and logical order
Asset approval	Every image or logo is author-approved, or omitted
Scope control	No analytics, form, CMS, tracker, or publication action was introduced
Handoff	Journal records the next content or style review

This verification card is useful even if the portfolio never becomes public. It teaches the central habit: public-facing output should have a traceable relationship to the facts and approvals behind it.

Next step

Chapter 6 shifts from static content to application state. You will build a React todo app that saves only to the browser's localStorage. The new challenge is not public claims; it is making state changes, persistence, and tests understandable across agent sessions.

Reader exercise: an honest one-page portfolio

Give ClineFlow three approved facts—who you are, one project you can describe truthfully, and one way readers may contact or learn more about you. It creates the source record and a content map before writing the one-page portfolio; you review the claims before approving the page.

Review the generated map for three risks: claims the source does not support, information you do not want public, and details that are missing but tempting to invent. Tell ClineFlow the corrections so it can update the approved record. Only then ask for a static structure.

When the first version is visible, perform the verification card yourself. Do not delegate the final judgment about your identity or work. You can ask the agent to identify possible contrast, hierarchy, or responsive-layout problems, but you decide whether the page accurately represents you.

For an extra practice round, ask the agent to propose a more energetic headline. Keep its proposal in the journal as a draft. Compare it with your approved bio. If it exaggerates, makes a promise, or uses a voice that is not yours, reject it and say why. That rejection is valuable context. It teaches the next agent what the site must preserve.

The finished learning project may be simple. Its success is not measured by a launch or a follower count. It is measured by whether you can return later, identify the approved source behind each claim, make a small change without losing the design's constraints, and leave a clear next step for future work.

A compact portfolio journal template

Use this template whenever you make a public-facing change:

```
---
type: Engineering Journal
```

```
title: Portfolio content revision
status: draft
---

# Goal

Add or revise one author-approved portfolio statement.

# Sources

- /knowledge/references/approved-bio.md
- /knowledge/references/approved-projects.md

# Constraint

Do not change any public claim not required for this revision.

# Proposed change

<Describe the smallest content or layout edit.>

# Verification

<Name the exact reference checked and the browser inspection performed.>

# Next step

<State the next safe author review.>
```

The `Sources` heading has a practical purpose. It reminds the next agent that it must trace the visible page back to approved material. It also makes a later correction easier. If you discover that an outcome sentence is too broad, you can find the source record, revise the sentence, and note the decision without guessing which conversation created it.

Separate visual work from factual work

An image, logo, or decorative visual can change the meaning of a page. Treat visual approval as a separate decision from content approval. A project card may have approved text but no approved image. In that case, use a clean text-first card. Do not ask an image model to invent a client logo, product screenshot, award badge, or professional portrait as a shortcut.

When an original visual is approved, record which file is approved, what it represents, and where it may appear. This protects the visual work from accidental replacement and helps a future agent

avoid treating an attractive generated asset as evidence of a real project.

The same discipline applies to the book cover later in this project: visual direction can be established now, but a final expressive visual is not substituted or released without explicit author approval.

Final portfolio question

Before ending work, ask: “Could a careful reader distinguish approved facts, author choices, and unfinished ideas?” If the answer is yes, the journal is doing its job. If the answer is no, add one short decision note before moving on. A transparent draft is stronger than a polished page whose provenance has disappeared.

Preserve that distinction.

Continue the chapter online

Build a truthful portfolio from approved source records.

Next: Ask ClineFlow to check the journal and plan the portfolio.

Open the companion lab →

CHAPTER 06

REACT LOCAL-FIRST TODO APP

Small tasks. Durable progress.

Reader outcome

Build a React/Vite todo app using localStorage, tests, and durable feature history.

Planned sections

- Define the local-first model.
- Ground the component map before editing.
- Implement, test, journal, and resume.

> **ClineFlow automation rule.** Once installed, ClineFlow inspects the starter app, generates and maintain its OKF context, journal decisions and verification, and leave the next handoff. You give the feature boundary, approval, and review —not files.

The result is a transparent, local, database-like memory layer alongside the React app. It lets a fresh chat or compatible agent recover the task model and test history without your retelling them. When a verified feature slice is ready, say **“please commit”**; ClineFlow carries the code, context update, and next safe step together.

Start with a small, inspectable boundary

A todo application seems ordinary, which makes it an excellent context exercise. It has visible state, a modest user interface, and decisions that accumulate quickly: What is a task? May it be empty? Does completing it remove it? What happens when the browser closes? An agent can generate a polished component in minutes while quietly choosing answers to all of those questions.

ClineFlow makes the choices readable before they become accidental product behavior.

This chapter builds a learning application with React and Vite. Tasks remain in the browser through `localStorage`; there is no account, server, synchronizing service, analytics, or real user data. “Local-first” here is deliberately modest: the first useful version works on one browser on one device, and the reader can inspect exactly where its data comes from. It is not a promise of backup, encryption, collaboration, or cross-device sync.

Open an empty `todo-lab` folder and ask ClineFlow to initialize the Vite app and its project memory with the following boundary. Do not create or maintain this tree manually:

```
todo-lab/
  knowledge/
    index.md
    journals/todo.md
    decisions/task-model.md
    decisions/local-data-boundary.md
    references/feature-brief.md
  src/
  tests/
  package.json
```

The generated context does not make React remember anything by itself. It gives a human and an agent a durable place to find the current product definition, decisions, and evidence from the previous session. State these two decisions in ordinary language; ClineFlow records them:

```
# Task model

A task has a stable id, a non-empty title, and a completed boolean.
The first version supports creating, toggling, and deleting one task at a time.
It does not contain due dates, accounts, sharing, tags, or notifications.
```

```
# Local data boundary

The app reads and writes only its own example task array in browser
localStorage.
It must not send tasks to a network service. Clearing browser storage may remove
the learning data; this limitation is shown in the interface or README.
```

These are product constraints, not incidental implementation notes. They let a non-developer ask useful questions as well: “Will my list leave this computer?” “What information is the agent allowed to see?” and “Which feature has not been approved yet?” A developer can translate the decisions into types and tests; a writer, lawyer, or project manager can still review their meaning.

Locate, understand, and introspect

Do not start with “build a complete todo app.” Ask the agent to inspect the existing project and knowledge files first. The sequence follows the grounded workflow: locate the relevant material, understand it, introspect about uncertainty, plan a bounded change, define success, ground the implementation, then execute and verify it.[36]

Use this first prompt after Vite has created the starter project:

```
Check the journal and help me understand the todo app. Do not edit anything.
Summarize the current behavior, identify any missing requirement that would
affect task persistence, and list the smallest
component and test changes needed for create, toggle, delete, and localStorage
restore. State what you cannot infer instead of inventing a requirement.
```

The request is intentionally review-first. It prevents the common failure in which an agent swaps the Vite setup, introduces a state library, or adds a remote database merely because those choices are familiar. If the repository already has tests, the agent should name them before proposing replacements. If there are no tests yet, it should say so and propose the narrowest first test.

Define success before the first edit

Record success conditions where the next session can find them:

- A reader can add a non-empty example task and see it immediately.
- A reader can mark one task complete and toggle it back.

[36] Hassan Uriostegui, “Stop Telling AI Agents What to Do.”

- A reader can delete one chosen task without changing another.
- Reloading the page restores the example tasks from this app's localStorage key.
- Malformed or absent stored data does not prevent the app from rendering.
- Tests cover the task behavior you claim is implemented.
- The project does not make a network request or suggest that browser storage is a secure backup.

These conditions distinguish a demonstrable first slice from a feature list. They also create a fair verification target for an agent: the change is done when observable behavior and tests agree, not when the component merely looks finished.

Plan, ground, execute, and verify

After the reconnaissance answer is reviewed, ask for a plan before code. A useful plan names each file, describes the state transition, and connects a test to a reader-visible result. It should be small enough that you can reject one part without throwing away the rest.

```
Check the journal and write a numbered plan only for this React todo app. For
each proposed change, name the file, the behavior it changes, the relevant
task-model or local-data constraint, and the verification step. Keep the first
slice to add, toggle, delete, load, and save. Do not add packages, routing,
accounts, cloud storage, styling systems, or features not named in the brief.
End with risks and questions that need a human answer.
```

Read the plan with the task model beside it. A reasonable first implementation often has one stateful application component, a small task-list display, an input and add button, and a persistence helper. The exact component names do not matter. The boundary matters: a UI component should not make an unexplained network request, and persistence code should not quietly redefine what a task is.

A narrow persistence rule

`localStorage` stores strings, so the app must serialize the task array when it saves and parse it when it loads. Treat browser storage as untrusted input. A previous development build, manual browser edit, or old data shape can leave a bad value behind. The first version need not solve every migration problem, but it should recover safely: if the expected array is absent or cannot be parsed, start with an empty example list and let the interface render.

State the rule to the agent so ClineFlow writes it in the feature brief: “Only an array of task-shaped records is accepted; unusable stored data falls back to an empty list.” That sentence gives the agent a testable requirement. It is more useful than “handle errors,” which can mean anything from hiding an exception to silently deleting user data.

Now ground the implementation request:

```
Implement the approved first slice exactly as planned. Preserve the Vite
configuration and existing dependencies unless the plan named a necessary
change. Keep task state in React, persist only the defined task array under one
clearly named localStorage key, and recover to an empty list when stored data is
missing or unusable. Do not add network calls, accounts, telemetry, deadlines,
tags, or bulk actions. After each logical change, run the existing relevant
tests and report the files changed and any assumption you had to make.
```

This prompt does not ask the agent to imitate a particular architecture. It asks it to honor a checked boundary. A beginner can review the final report against the decision files; an experienced React developer can inspect whether the state and persistence effect are placed sensibly.

Test the behavior, not the library

Tests should express the contract a reader can recognize. Depending on the test tools already in the starter, ask the agent to cover these cases:

1. Submitting visible non-empty text adds a task.
2. Submitting blank or whitespace-only text does not add a task.
3. Toggling one task changes only that task’s completed state.

4. Deleting one task leaves the other rendered tasks intact.
5. A stored valid task array is displayed after startup.
6. Missing or malformed stored data leaves a usable empty interface.

Manual browser review complements the tests. Add two sample tasks, complete one, reload, and confirm that the state returns. Then remove the app's storage key in browser developer tools and reload again. The page should still be usable. Do not test this with real private tasks; the chapter's project is intentionally a synthetic learning environment.

Journal the state of the project

Code explains the current mechanism. A journal explains why the mechanism is there, what was checked, and what work is still safe to do. After the first slice, write a record that another agent can use without reconstructing your session from component names:

```
---
type: Engineering Journal
title: First local todo slice
status: active
---

# Goal

Create, toggle, delete, and restore example tasks without a network service.

# Decisions followed

- A task has id, non-empty title, and completed state.
- Browser localStorage holds only the app\u2019s example task array.
- Invalid stored data falls back to an empty list.

# Verification

- Ran the relevant test command successfully.
- Added two sample tasks, toggled one, reloaded, and checked restoration.
- Removed the storage key and confirmed the empty state still renders.

# Open questions

No decision exists yet for editing titles, filtering, due dates, or backup.

# Next step
```

```
Inspect the current test and component structure before proposing one small
accessibility or empty-state improvement.
```

This is not administrative decoration. The next request can point to the journal, so it begins with confirmed behavior and unresolved boundaries rather than an optimistic summary such as “the todo app is done.” That distinction is how a short context window becomes durable project context: the important information is stored outside the conversation and can be located again.

Continue with one observable improvement

Choose the next feature only after locating the current code and journal. An empty state is a good second slice because it adds a reader-visible result without changing the task model. It might say that there are no example tasks yet and direct the reader to the input. It must not imply that tasks were backed up, synchronized, or sent to another service.

```
Check the journal and inspect the todo app. Do not edit yet. Propose the
smallest accessible empty-state improvement. Explain its
effect on existing success criteria and tests. Do not introduce new task fields,
dependencies, remote services, or unrelated styling changes.
```

After approving the plan, use a similarly narrow execution request: implement only the approved empty state, retain all existing create/toggle/delete and persistence behavior, add or update the relevant test, run the test command, and append a concise journal result. If the agent discovers that an earlier assumption was wrong, it should stop and report it rather than silently redesigning the data model.

Accessibility is part of the feature

The todo app should work without a pointer. Label the input, give the add action an understandable name, use a real checkbox or clearly equivalent control for completion, and give destructive

actions a specific accessible name such as “Delete Buy tea.” Test with the keyboard: can you reach the input, add button, each completion control, and each delete action in a sensible order? When a new task is added, can the reader tell that the result occurred without relying only on color?

Ask the agent to report what it checked, but perform a human review as well. Automated tests can confirm that elements exist; they do not guarantee that the interaction feels understandable. Record a concrete result, such as “Tab moved from the title input to Add task, then through each checkbox and its paired delete button; completed text remains readable without color.”

Common failures

- **Feature drift.** A request for an empty state becomes routing, themes, user accounts, or a backend. Restore the boundary and request one behavior at a time.
- **Storage treated as a vault.** Browser storage is presented as encrypted or as a reliable backup. Remove that language; the lab makes no such promise.
- **State mutation.** Toggling one task changes another or prevents React from updating. Write a behavior test that isolates the selected task.
- **Invisible failure.** Bad stored data crashes the page or disappears without a record. Use the documented empty fallback and journal the condition.
- **Tests as theater.** A test only asserts that a component mounted. Prefer a user action and an observable result.

When a failure occurs, let ClineFlow add the smallest fact to the journal: what the reader did, what happened, what decision applies, and the next safe check. This makes the failure useful context rather than a story that vanishes with the chat.

A feature-evolution exercise

Once the first slice is stable, practice extending it without losing its history. Choose one deliberately small feature: a filter that shows all, active, or completed tasks. Before requesting code, decide whether filtering changes the stored task model. In this version it does not. Filter selection is a temporary view choice, not another field written into every task record.

Put that distinction in a decision note, then use the full prompt loop:

```
Check the journal and inspect the todo app. Understand the current task model
and verify whether filter state belongs in persisted task data. Introspect about
missing requirements.
Plan the smallest all/active/completed filter. Define success criteria. Do not
edit until the plan is approved.
```

After review, ground execution with the approved decision and ask the agent to implement only the filter controls and related tests. Verify that adding, toggling, deleting, reloading, and malformed-storage recovery still work. Then journal the result and name a next step. This repeated loop is the chapter's real deliverable. The todo interface is intentionally small so that the reader can see the whole chain from source decision to changed behavior.

Do not confuse preserved context with refusal to change. Durable context makes change safer because the next request can tell which constraints are still true, which behavior was already verified, and which question is genuinely new. If you later decide to add editing, due dates, or cloud sync, make each one a new decision. Describe its data implications, privacy implications, and verification criteria before an agent writes it.

Final verification card

Check	Evidence to inspect
Task contract	`task-model.md` matches the fields the UI creates and displays
Local boundary	Source and network panel show no service integration; README explains browser-only storage
Behavior	Add, toggle, delete, reload, and invalid-storage fallback were tested
Accessibility	Controls have useful labels and a keyboard review result is in the journal
Tests	Relevant tests ran and cover observable task behavior
Handoff	Journal names confirmed scope, open questions, and the smallest next change

The network panel check deserves care. Its purpose is to detect an accidental integration, not to prove a privacy guarantee. A browser, development tool, or dependency may have its own traffic. The project's source, package choices, and explicit no-service boundary are the primary evidence for what this learning app intends to do. If something unexpected appears, stop and inspect it before placing real information in the app.

Prompt card: resume a paused session

```
Check the journal and continue from the last verified step. First report
confirmed behavior, open questions, and the last verification result. Then
propose one
small next step with success criteria. Do not edit or broaden scope until I
approve the plan.
```

This is useful whether the agent is Cline, Codex, Cursor, Copilot, or a local tool. The important context is in the repository, not in a vendor-specific chat history. A new agent may use different words, but it can locate the same facts, inspect the same code, and leave a journal the next agent can use.

Next step

Chapter 7 takes the same local-first discipline to an iOS SwiftUI todo app. The technology changes, but the workflow does not: define the on-device boundary, locate the project's model and journal, make one state change at a time, verify it in the simulator, and leave a durable record for the next iteration.

A durable project note states what changed and what remained deliberately unchanged, so future work begins from evidence rather than assumption alone.

Context survives change.

A regression ritual for local-first state

Every new view feature can accidentally break the state behavior already earned. Before merging a filter, empty-state, or accessibility improvement, run the same short regression ritual. It is intentionally independent of a specific test library so a beginner can perform it in a browser and a developer can automate parts of it in the project's existing test setup.

Start from a known state. Clear only this lab's localStorage key, reload, and confirm the documented initial list or empty state appears. Add two fictional tasks with distinct titles. Toggle the first task, reload, and verify that its completed state returns while the second task remains unchanged. Delete the second task, reload again, and verify that the first remains. Finally replace the storage value with unusable example data and check that the interface shows the safe empty result instead of crashing.

These actions map directly to the decisions and success criteria already stored in the knowledge bundle. When the outcome differs, resist the urge to ask the agent for a broad fix. Journal the smallest failing action and its visible result, then use a locate-first prompt:

```
Check the journal and inspect the todo app. Do not edit. Compare the recorded
regression failure with the intended behavior. Identify the smallest
likely code and test area to inspect, state uncertainties, and propose a
bounded repair plan. Do not add a package, service, account, persistence
feature, or unrelated redesign.
```

The result should be a plan, not a guess disguised as a patch. For example, a bad-data crash may point to the loading helper and its test; a filter that deletes hidden tasks may point to the update path. The agent should name the decision it is preserving. If it cannot identify a controlling decision, that is a sign to pause and let the human clarify the product rule first.

Useful tests for a changing app

As the app grows, retain a small set of tests that protect its core contract:

- Reading a valid serialized array returns task-shaped records.
- Reading absent, malformed, or wrong-shaped data returns the documented safe fallback.
- Adding trimmed non-empty input creates one task; blank input creates none.
- Toggling a selected id changes that task alone.
- Deleting a selected id removes that task alone.
- A view filter changes which rows display without altering stored task records.

Do not test implementation trivia merely because it is easy to assert. A test that names an internal state variable will break during a harmless refactor and may say nothing about what a reader sees. Prefer inputs and outcomes that match the task model and journal. The point is not a high test-count number. It is a reliable check that the behavior the chapter promised still exists.

A recovery journal entry

```
# Regression result

Observed: After selecting Active, deleting the visible task left completed tasks
unchanged and local storage retained their original records.

Evidence: Browser regression ritual and the selected task behavior test passed.

Boundary: No service, account, telemetry, or data-model field was added.

Next step: Inspect keyboard focus after a task is deleted.
```

This small entry prevents a future filter enhancement from re-litigating whether the filter belongs in persisted data. It also makes a failed test useful: the next agent begins with the observed behavior and a decision boundary, not an unexplained red indicator.

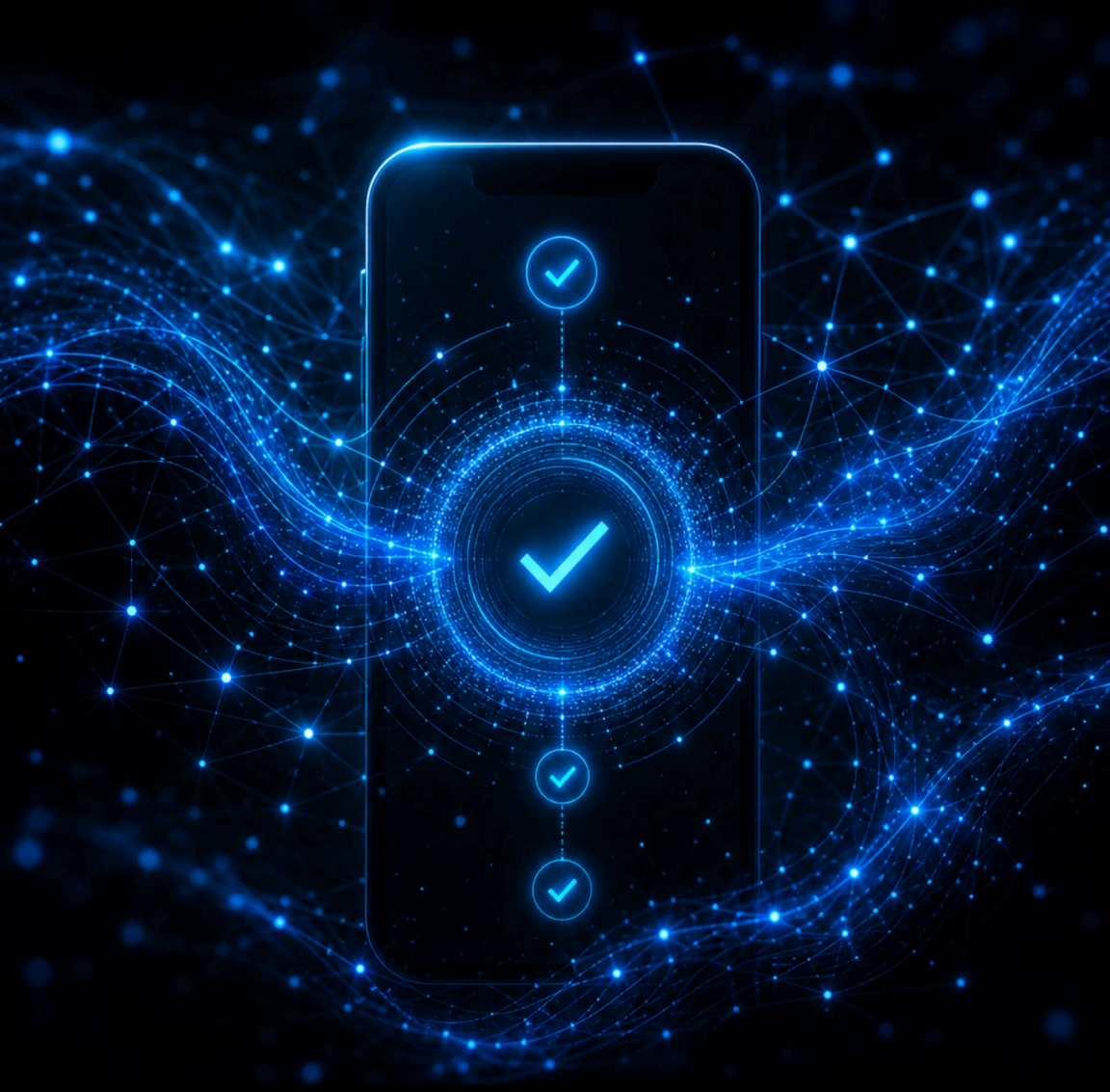
Keep the ritual proportionate. A two-minute repeatable check performed after each small change is more protective than an elaborate test plan nobody runs. When it finds a discrepancy, preserve the observation first and automate the case when it will protect the next expected change.

Continue the chapter online

Build and regression-check a browser-local todo app.

Next: Ask ClineFlow to check the journal and plan the todo app.

Open the companion lab →



CHAPTER 07

iOS LOCAL-FIRST TODO APP

Keep the task on the device.

Reader outcome

Build a SwiftUI todo app with on-device persistence and a clear record of product decisions.

Planned sections

- Project setup and app states.
- Local persistence and verification.
- Prompting an agent through an iOS change safely.

> **ClineFlow automation rule.** After ClineFlow is installed, its workflow directs the agent to establish and update the OKF project memory, decisions, journal, checks, and handoff. You supply the app boundary, approval, and review—not the supporting files.

That local, readable OKF layer is the app's transparent project memory, not another documentation job. A fresh VS Code/Cline chat, Codex session, or cloud agent can recover the same on-device boundary from it. After a verified slice, say **"please commit"** and ClineFlow preserves the code, check, and handoff as one durable update.

Make the on-device boundary explicit

An iOS todo app is a useful companion to the browser project in Chapter 6. The surface changes—SwiftUI views, an Xcode project, and a simulator—but the context problem is familiar. An agent can produce a list interface quickly and quietly decide where task data lives, what happens after a relaunch, and whether the app talks to a service. State those answers to ClineFlow; it records them in the project automatically.

This learning app keeps fictional tasks on the device. It has no sign-in, sharing, cloud synchronization, analytics, push notifications, or real client matter. On-device persistence means that the first version stores the small task list locally for the app; it does not mean the data is encrypted, backed up, portable to another device, or suitable for sensitive information. If your work involves confidential material, use only synthetic examples for this lab and follow the relevant organizational and professional requirements outside the lab.

Create a new SwiftUI app in Xcode, then ask ClineFlow to establish this structure alongside it. Do not construct or maintain the knowledge files by hand:

```
SwiftTodoLab/
  knowledge/
    index.md
    decisions/task-model.md
    decisions/on-device-boundary.md
    journals/ios-todo.md
    references/feature-brief.md
  SwiftTodoLab/
    TodoItem.swift
    TodoStore.swift
    ContentView.swift
  SwiftTodoLabTests/
```

The names are examples, not a required architecture. What matters is that the model, persistence choice, current state, and verification record can be found without relying on a previous chat. State a small model decision in plain language and let ClineFlow record it:

```
# Task model

A task has a stable id, a non-empty title, and a completed state.
The first release creates, toggles, and deletes tasks. Editing titles, dates,
tags, accounts, sharing, and synchronization are out of scope.
```

Then add the on-device boundary:

```
# On-device boundary

This learning app persists only fictional task records locally for this app.
```

```
It makes no network request and includes no account or cloud service. Local
storage is not described as a secure archive, backup, or legal record system.
```

These decisions are approachable for every reader. A developer can map them to a Swift type and persistence code. A lawyer can recognize that no client data belongs in the example. A creative can decide which kind of private work should remain outside a sample project. The agent has fewer reasons to "helpfully" add a platform service that the author never intended to use.

Locate before you change Xcode files

Create a new SwiftUI app in Xcode using the standard template, then pause before asking an agent to restructure it. Xcode projects contain configuration and generated metadata that are easy to disturb when a request is broad. First ask the agent to check the journal, understand the app's current model and views, test target, and any existing persistence code. It should describe what it found without editing.

```
Check the journal and help me understand the iOS todo app. Do not edit.
Summarize current behavior, identify what is absent, and
state the smallest plan for fictional task create, toggle, delete, and local
restore. Name any uncertainty rather than inventing an iCloud, account, or
network requirement.
```

This is the same locate → understand → introspect stage used throughout the book.[37] The platform is different, but the durable context is still the knowledge folder, source tree, tests, and journal. A later agent can inspect those records even if it was not present when the app was created.

First-slice success conditions

- A reader can create a non-empty fictional task.
- A reader can toggle and delete one selected task.

[37] Hassan Uriostegui, "Stop Telling AI Agents What to Do."

- Relaunching the app restores valid local example data.
- Missing or unusable persisted data leaves a usable empty state.
- The UI offers understandable labels and does not rely only on color.
- Tests and simulator checks cover the behavior you say is present.
- The project remains free of network, account, cloud, and sensitive-data features.

Plan the model and store before the view

Once the reconnaissance response is approved, ask for a plan that separates the task model, local store, screen state, and tests. SwiftUI makes it tempting to put everything in a single view because a tiny prototype can work that way. A small store is clearer for this lab: it gives persistence one home and makes the screen responsible for showing and requesting changes rather than silently deciding how data is saved.

```
Check the journal and write a plan only for this SwiftUI todo app. For each
change, name the Swift file or test file, the behavior it owns,
the stored-data effect, and how it will be verified in the simulator or tests.
Keep the plan limited to a TodoItem model, a local TodoStore, a SwiftUI list,
and create/toggle/delete actions. Do not modify project settings, add packages,
iCloud, accounts, networking, notifications, or unapproved fields.
```

Review the plan against the decisions. A suitable `TodoItem` might have an id, title, and completion flag. The store can keep an in-memory array and encode it to a local representation after a change. The exact persistence API can vary by the reader's platform and project template, so the agent should identify the one already chosen rather than claiming there is only one correct option. The book's durable-context lesson is not "always use this storage API." It is "make the storage choice explicit, small, and verifiable."

Defend against stored-data surprises

Local data can be absent, malformed, or left behind by an earlier experiment. Treat it as input to validate rather than as an unquestioned source of truth. The first version needs a simple recovery rule: if decoding fails or the expected collection is unavailable, begin with an empty list and keep the interface usable. Do not erase a value merely because the app has not yet explained what went wrong; record the behavior in the journal and decide later whether a recovery message is appropriate.

Tell ClineFlow to add a clear sentence to the feature brief: “Valid local task records restore at launch; unavailable or unusable records produce an empty task list without a crash.” This is specific enough for a test and for a simulator check. It avoids the vague request “handle persistence errors,” which lets an agent choose an unreviewed failure policy.

Ground the first implementation

After approving the plan, give the agent a bounded execution request:

```
Implement the approved first slice. Preserve the existing Xcode configuration
and dependencies. Keep fictional task state in the local TodoStore; encode only
the id, title, and completed fields to the approved on-device storage. Restore
valid records on launch and use an empty list for unavailable or unusable data.
Build a SwiftUI interface that creates a non-empty task, toggles one task, and
deletes one task. Do not add networking, sign-in, cloud sync, telemetry, dates,
tags, notifications, or unrelated visual redesign. Report changed files,
assumptions, and the exact test or simulator checks you ran.
```

The words “preserve” and “report” matter. They prevent a harmless todo feature from becoming a project-file rewrite or a platform-integration experiment. They also make review possible after the chat ends: the journal can name each changed file and distinguish a confirmed behavior from an assumption needing a human decision.

Verify with tests and the simulator

Use tests for model and store behavior when the project target supports them. At minimum, demonstrate that a valid task record round-trips through the local representation and that an invalid value produces the documented empty result. Then use the iOS simulator to add two fictional tasks, toggle one, delete the other, close and relaunch the app, and check that the remaining valid state restores. Clear the app's local learning data and relaunch once more; the empty state should be understandable and usable.

Simulator review catches what a model test cannot: truncated text, an unclear add action, an unreachable delete control, or a completion state that relies only on color. Record the device or simulator configuration used, the actions performed, the result, and anything still uncertain. That record is the starting point for the next agent session—not a claim that the app is finished.

Leave an iOS-focused journal

At the end of the first working session, let ClineFlow write the journal that connects the model, store, screen, tests, and simulator evidence. Avoid a vague status such as "SwiftUI todo complete." A future agent needs the exact behavior that was checked and the decisions it must not silently overturn.

```
---
type: Engineering Journal
title: First SwiftUI todo slice
status: active
---

# Goal

Create, toggle, delete, and restore fictional tasks on this device.

# Decisions followed

- Task fields are id, non-empty title, and completed.
- The app stores only fictional tasks locally.
- Invalid or missing local data becomes an empty usable list.
```

```
# Verification

- Ran the relevant model or store tests.
- In the simulator, added two tasks, toggled one, deleted one, and relaunched.
- Cleared local learning data and confirmed the empty state still worked.

# Open

No decision exists for editing, dates, accounts, sharing, backup, or cloud sync.

# Next step

Inspect current views and tests before proposing one accessible empty-state or
filter improvement.
```

This journal is portable context. It lets an agent that did not build the first slice identify the app's promises, evidence, and open edges. It also gives a human reviewer a place to correct a mistaken assumption before it becomes code.

Improve one state without broadening the app

An empty state is a good follow-on change. It is visible, useful, and does not need a new task field. It can tell a reader that there are no fictional tasks yet and direct attention to the add control. It should not claim that a list was saved securely or invite a reader to add private client, personal, or work data.

Start again with a review-first request:

```
Check the journal and inspect the iOS todo app. Do not edit. Explain the
smallest accessible
empty-state improvement, its effect on current behavior, and the test or
simulator checks needed. Do not add a new data field, package, account, cloud
feature, network request, or unrelated styling change.
```

Only after approving the plan should the agent make the small change. Ask it to retain create, toggle, delete, and restore behavior, update the relevant test where appropriate, run the selected check, and append a journal note. If it finds that an empty state needs a product decision—perhaps whether to show sample tasks—it should stop for that answer rather than inventing one.

Accessibility review in the simulator

Use clear text labels for adding a task and for deleting a specific task. A completion control should have a recognizable selected state beyond color alone. Check Dynamic Type or a larger accessibility text setting: do task titles wrap without hiding the delete action? Navigate with the available simulator and accessibility tools, then record a concrete result. For example: "At large text, the title wraps and Delete remains readable; the empty state describes the next action; completed rows remain distinguishable without color."

An agent can suggest an accessibility improvement, but it cannot replace the reader's observation of the running interface. Keep the check in the journal with the device configuration used. That is stronger context than a general statement that the app is accessible.

Common failures

- **Cloud creep.** A local todo request becomes an iCloud, account, or sync feature. Return to the on-device boundary and make a separate future decision.
- **Project-file churn.** The agent rewrites Xcode configuration for a view change. Stop, inspect the diff, and restore the planned file boundary before continuing.
- **Unclear recovery.** A bad stored value crashes the app or is silently treated as a valid task. Test the documented empty fallback.
- **Simulator-only confidence.** The app looks right once but has no store/model check. Add a narrow behavior test.
- **Color-only completion.** A completed task is communicated only by a tint. Preserve readable text and an understandable control state.

A controlled feature-evolution exercise

When create, toggle, delete, restore, and the empty state are confirmed, try a view-only filter: all tasks, active tasks, or completed tasks. The exercise is small by design. It teaches that a changed interface does not automatically justify a changed stored model. In this version, the selected filter is a temporary screen choice; it does not add a field to each `TodoItem` and it does not expand the on-device boundary.

First tell ClineFlow to create a decision note stating exactly that. Then request the familiar automated loop rather than asking for "filters" in a single breath:

```
Check the journal and inspect the iOS todo app. Understand the current storage
model.
Introspect about whether filter selection needs to be persisted. Propose a
small all/active/completed filter plan with success criteria and verification
steps. Do not edit until I approve it. Do not add new task fields, packages,
accounts, network calls, cloud sync, or unrelated UI changes.
```

The expected answer should identify which existing behavior must remain true. Filtering must not change a task's completion status, delete a hidden task, or interfere with local restoration. The agent should propose a test or simulator scenario that toggles a task while filtered, changes the filter, and confirms that the underlying list is unchanged. If it proposes persisting filter state, ask why and make a separate author decision before proceeding.

After approval, ground the execution request in that plan. Require a narrow diff, an explicit list of changed files, a test run where available, and a simulator check. The journal handoff might say: "The selected filter changes only what the list displays; task records retain their original three fields. Verified by toggling an active task, switching to Completed, relaunching, and checking that restored task state—not filter choice—matches the stored model."

Verification card

Check	Evidence to inspect
Model boundary	`TodoItem` and `task-model.md` agree on id, title, and completed only
Storage boundary	Store and decision record show local fictional data only; no service configuration exists
Core behavior	Simulator result covers create, toggle, delete, relaunch, and empty fallback
Accessibility	Large-text and understandable-control observations are written in the journal
Tests	Model/store tests cover a valid restore and invalid-data fallback
Handoff	Journal states the last verified change, remaining questions, and one safe next step

The verification card does not certify an app for every device or use case. It gives a reader an honest, repeatable review for this learning slice. When the facts change—for example, a future app adds a service, a new storage policy, or real data—the decision record and verification plan must change too.

Prompt card: resume safely

```
Check the journal and continue from the last verified step. First state
confirmed behavior, the exact last
verification result, and open questions. Then propose the smallest next change
with success criteria. Do not edit, add a dependency, change project settings,
or broaden the data boundary until I approve the plan.
```

This request works with a cloud agent or a local agent because the important context lives in the project. The agent's context

window may be limited; its ability to locate current decisions, code, tests, and journal entries is what makes a multi-session app manageable.

Next step

Chapter 8 moves from application state to story state. You will create a fiction project that keeps a world bible, character records, plot decisions, and continuity checks. The same rule applies: an agent may help explore prose, but the human author controls the canon and leaves a durable decision trail.

Review each change deliberately.

A simulator regression routine

The simulator is most useful when it repeats a known sequence after a small change. Do not rely on a single launch that happens to show the intended list. Begin with the app's documented initial state. Add two fictional tasks whose titles make them easy to distinguish. Toggle the first, relaunch, and confirm that the completed state returns. Delete the second, relaunch again, and verify that the first remains. Then clear the app's local learning data, launch once more, and confirm that the empty state is readable and the add control still works.

Let ClineFlow write the test conditions in the journal before you run them. This helps a future agent understand whether a filter, empty state, or view refactor changed the model/store contract. It also avoids treating a simulator screenshot as proof that persistence is correct. The evidence is the sequence of actions, configuration, and observed result.

```
# Simulator regression record

Device: <simulator model and OS version>
Initial state: <empty list or named fictional samples>
```

```
Actions: add two tasks; toggle first; relaunch; delete second; relaunch; clear
data
Observed result: <state after each action>
Accessibility check: <large-text/control-label observation>
Decision: <confirmed, investigate, or revise a stated rule>
```

If one action fails, return to project discovery before asking for a repair. The store may be loading invalid data, a view may be updating a copy rather than the stored item, or an unreviewed filter may be changing the wrong collection. Use a bounded recovery prompt:

```
Check the journal and inspect the iOS todo app. Do not edit. Compare the
recorded simulator
failure with the intended behavior. Identify the smallest model, store, view, or
test area to inspect and propose a repair plan with a simulator regression step.
Do not add packages, change Xcode configuration, add networking, account,
analytics, cloud sync, or new task fields.
```

The agent should say what it cannot establish. A failure after relaunch might be caused by persistence, but it could also be caused by the test setup or an unclear observation. The journal lets the human distinguish evidence from a hunch before making a change.

Protect the model as features grow

When adding a view-only filter, keep the model test suite narrow and durable. Test that a task encodes and decodes with the defined id, title, and completion fields. Test that a missing or unusable local value produces the documented empty collection. Test that toggling a selected task changes only that task and that deleting it does not alter another. Filter tests should assert which rows are visible without requiring a new persisted property.

This protects a fundamental SwiftUI boundary: the screen may have temporary state, while the store owns the durable fictional task records. It also prevents an agent from putting a convenience flag into each task merely because the UI currently needs it. If the project later needs a persisted preference, make a new decision

that explains why, what it stores, and how a reader can verify its behavior after a relaunch.

A concise recovery handoff

```
# Regression result

Observed: Filtered view displayed correctly; relaunch restored tasks, not the
temporary filter selection.

Evidence: Model/store test passed; simulator sequence completed on the named
device configuration.

Boundary: No account, service, cloud feature, real data, or project setting was
added.

Next step: Review focus order and large-text layout after deleting a task.
```

Like the rest of ClineFlow, this record keeps the next change reviewable. It does not claim that one simulator run proves universal behavior. It says what was actually seen, which constraints still hold, and what remains to check.

Continue the chapter online

Build and inspect a fictional on-device SwiftUI todo app.

Next: Ask ClineFlow to check the journal and plan the iOS todo app.

Open the companion lab →

CHAPTER 08

FICTION PROJECT MEMORY

Keep the canon human.

Reader outcome

Use an open story bible to protect continuity while keeping the human author in charge.

Planned sections

- World, character, and plot records.
- Draft status, source boundaries, and continuity review.
- Prompt patterns for revision without surrendering authorship.

> **ClineFlow automation rule.** ClineFlow creates and maintains the story-memory records, journal, continuity checks, and next-session handoff after installation. You supply the story intent, canon decisions, approval, and editorial judgment—not the record-keeping work.

This makes the story bible a transparent, local, database-like memory layer: inspectable OKF records that survive a new chat or a change of agent without forcing the author to rebuild the context. After a verified editorial slice, say **“please commit.”** ClineFlow preserves the draft change, canon decision, continuity check, and next handoff together while the author remains in charge.

Treat the story bible as working memory, not as a command to write

Fiction is full of long-range constraints: a character’s fear changes after a specific scene, a city has rules the plot must honor, an object is already in someone’s pocket, and a mystery must not reveal an answer too early. A chat can hold some of this information for a while, but a story project outlives any one conversation.

A durable story memory lets an author and an agent locate the current canon without pretending that a generated paragraph is canon merely because it sounds confident.

The author remains the authority. The agent can summarize, search for possible contradictions, propose questions, or draft an explicitly requested experiment. It cannot decide what the story means, make a character's voice "better," or silently turn a speculative option into a settled fact. Those boundaries make the project useful for writers and equally legible to an editor, collaborator, or future self.

Ask ClineFlow to create a small story-memory bundle beside the manuscript. You give it the story goal and author boundaries; it creates and maintains the structure:

```
novel-lab/
  knowledge/
    index.md
    world/places.md
    characters/mara-vale.md
    plot/sequence.md
    decisions/canon-policy.md
    decisions/spoiler-boundary.md
    journals/story.md
    references/author-notes.md
  drafts/
    chapter-01.md
```

You do not need a massive encyclopedia. Begin with the facts that later scenes are likely to depend on: a character's role and current goal, important relationships, the world rules that affect cause and effect, the status of major plot questions, and the current draft boundary. Keep each record short enough to review. Link from the index so an agent can start at the project map rather than searching blindly through prose.

A canon policy in plain language

Tell ClineFlow to create this explicit decision before an agent sees the draft:

```
# Canon policy

Only material marked Canon in an author-approved record controls the story.
Ideas, alternate scenes, agent suggestions, and unfinished draft text are not
canon unless the author records a decision. When sources disagree, flag the
conflict; do not resolve it by inventing a fact.
```

This one note solves a recurring creative-agent problem. Without it, an agent may read an abandoned passage, mistake it for a settled event, and build later prose on that error. With it, the agent can say, “The draft and character record disagree; which source should control?” That question protects the writer’s ability to change their mind.

Add a spoiler boundary as well. For a reader-facing synopsis or a chapter-level task, identify which future events the agent may inspect and which remain withheld. A detective novel, for example, may have a private solution record and a public chapter brief that reveals only what the current point-of-view character knows. The boundary is not secrecy theater; it prevents a continuity review from accidentally writing future knowledge into an earlier scene.

Locate, understand, and ask questions first

Before asking for dialogue, outline, or revision, require an inspection pass:

```
Check the journal and help me understand the requested story task. Do not edit.
Summarize confirmed canon relevant to this task,
list conflicts or missing facts, and identify information that is outside the
spoiler boundary. Ask questions where an author decision is needed. Do not turn
an inference, draft experiment, or agent suggestion into canon.
```

This follows the same grounded sequence used throughout the book: locate, understand, introspect, plan, define success, ground, execute, verify, journal, and continue.[38] In a fiction project, “introspect” means distinguishing facts from assumptions

[38] Hassan Uriostegui, “Stop Telling AI Agents What to Do.”

and noticing when the prompt asks the agent to decide something only the author can decide.

Define a safe outcome

For a first continuity task, success might mean: list only contradictions that can be traced to specific records; quote or name the conflicting locations; preserve the author's point of view; make no draft edit; and leave unresolved questions visible. This is more valuable than a generic request to "make the chapter consistent." It gives the author a reviewable report before prose changes begin.

Run a continuity review before a revision

The first execution request should produce a report, not replacement prose. A report makes the agent's reasoning inspectable and gives the author a chance to reject a false conflict. It also protects an unfinished draft: a scene can be awkward, intentionally ambiguous, or marked as an experiment without becoming a defect for an agent to repair.

```
Using only the approved sources inside the task\u2019s spoiler boundary, review
the
requested draft chapter for possible continuity conflicts. Do not edit the
draft. For each finding, name the draft location, the controlling canon record,
the apparent conflict, and confidence level. Separate confirmed contradictions,
questions for the author, and optional craft observations. Do not reveal or
infer withheld plot information. End with a smallest revision plan only if the
author approves a specific finding.
```

The distinction between a contradiction and a craft observation matters. "Mara says the moon is blue although `world/places.md` says it is silver" is a traceable continuity question. "The scene needs more tension" is a subjective reaction. Both can be useful, but they deserve different labels. The first can be resolved through a record or author decision; the second is an invitation for the writer to decide whether a change serves the story.

Make a revision request narrow

After the author approves one specific action, ground the next prompt in that decision. For example, the author may decide that the draft's blue moon is an intentional illusion and let ClineFlow update the world record. Or the author may decide that the scene must use silver. Only then ask the agent to make a small alteration:

```
The author has approved this decision: <paste the decision and controlling
source path>. Revise only the named paragraph in `drafts/chapter-01.md` to honor
that decision. Preserve point of view, tense, scene outcome, and all information
inside the spoiler boundary. Do not create new canon, summarize later events,
or revise unrelated language. Report the exact edit, source used, and any
remaining uncertainty for author review.
```

The prompt does not require the author to accept the wording the agent returns. Treat it as a draft proposal. Read it in context and compare it with the character and plot records. Keep, revise, or reject it. If you accept a new story fact, record that fact separately; accepting a sentence for rhythm is not always accepting every implication it contains.

Journal a creative decision, not just a generated result

Use a story journal to distinguish the agent's work from the author's choices:

```
---
type: Story Journal
title: Chapter 1 continuity review
status: active
---

# Task

Review the observatory scene against current canon without exposing the
solution.

# Sources inspected
```

```
- /knowledge/characters/mara-vale.md
- /knowledge/world/places.md
- /knowledge/plot/sequence.md
- /drafts/chapter-01.md

# Findings

The moon-color reference conflicts with the current world record. The motive
for Mara\u2019s hesitation is not settled canon and needs author direction.

# Author decision

Keep the blue image as a deliberate illusion. Add that exception to the world
record; do not reveal why it occurs yet.

# Verification

Re-read the paragraph, the updated world record, and the point-of-view boundary.

# Next step

Inspect the next scene for only character-goal continuity.
```

This record is intentionally more useful than “agent fixed continuity.” It says which conflict was real, who resolved it, which fact became canon, and what future work must avoid revealing. A new agent can locate this decision before suggesting a later chapter. An editor can see that a seeming inconsistency is a deliberate effect rather than an oversight.

Verify with a human read

After any accepted change, read the surrounding scene as a reader, not as a database. Does the new wording still sound like the viewpoint character? Does it preserve pacing and ambiguity? Did it introduce a detail that requires a new canon entry? Search the relevant character, world, and plot records for the changed terms. Then let ClineFlow add a concrete journal result: “Blue illusion retained; Mara has no privileged knowledge of its cause; scene outcome unchanged.”

This human pass is essential. Continuity is not the same as good fiction, and a consistent story can still lose its voice. ClineFlow gives the author durable memory; it does not delegate authorship.

Use records for continuity checks, not automatic correction

As the project grows, repeatable checks help you find places worth reading. They should create a review queue, not edit the novel. A simple character record can list a current goal, knowledge boundary, relationship status, and physical constraints. A plot sequence can list scene order, point of view, and events that have reached canon. An agent can compare those structured notes with a requested draft and point to possible mismatches. The author decides whether a mismatch is an error, a deliberate reveal, or an outdated record.

Use this prompt for a focused check:

```
Read the approved character record, plot sequence, spoiler boundary, active
story journal, and the named draft scene. Do not edit. Make a continuity review
table with: source record, draft location, possible mismatch, confidence, and
question for the author. Include only information permitted by the spoiler
boundary. Do not propose new canon or repair prose.
```

A table makes uncertainty visible. “Character knows a harbor map in the draft; the current chapter brief says she has not seen it” is a question. “Character is wrong” is not. This wording leaves room for the writer to say that the map is a dream, the brief needs revision, or the scene needs a rewrite. The method supports discovery without turning the agent into a continuity judge.

Common failures

- **Draft becomes canon.** An abandoned scene is treated as proof of a fact. Keep draft status explicit and require author approval for canon changes.
- **Suggestion becomes decision.** An agent offers a twist, then later treats it as established. Store suggestions separately from decisions.

- **Spoiler leakage.** A review imports the solution or a later revelation into an earlier viewpoint. Narrow the sources and restate the boundary in the prompt.
- **Style overwrite.** A “continuity fix” rewrites a page in a generic voice. Limit the edit to a named paragraph and review it in context.
- **False precision.** The agent labels a subjective pacing reaction a contradiction. Separate factual conflicts from craft observations.
- **Orphaned change.** The author accepts a new fact but the agent has not yet captured it. Ask ClineFlow to add the smallest canon decision before the next task.

When a failure occurs, preserve the useful fact in the journal: the source consulted, the incorrect assumption, the author’s resolution, and the next check. The goal is not to produce a perfect database. It is to avoid repeatedly paying the cost of rediscovering the same story decision.

A compact story-session ritual

Before a session, say “check the journal and help me continue this story.” Ask the agent to summarize constraints and questions. Approve a plan. Define success in terms of a report, a limited edit, or an experimental alternative. After the work, compare the result with the source records, decide what is canon, update only the required record, and journal the next step automatically.

```
Locate \u2192 understand \u2192 introspect \u2192 plan \u2192 define success
\u2192 ground \u2192 execute \u2192
verify \u2192 journal \u2192 continue.
```

The ritual may feel slower than a broad “write the next chapter” request. In practice, it preserves the choices that make a long work recognizably yours. You can still request play: ask for three non-canon scene possibilities, a list of sensory images, or questions a skeptical reader might ask. Mark those outputs as experiments. Promote one only through an author decision.

Final fiction verification card

Check	Evidence
Canon source	Relevant world, character, and plot records are named in the task
Spoiler control	Sources and output stay within the stated viewpoint boundary
Author control	New facts and accepted revisions have an author decision
Continuity	Findings identify draft locations and controlling records
Voice	Author re-read the revised passage in scene context
Handoff	Journal records confirmed canon, unresolved questions, and one next step

Next step

Chapter 9 applies the same discipline to a synthetic legal training matter. There, provenance and human review are even more important: the assistant can organize supplied material, but it must not provide legal advice, replace professional judgment, or receive real client information.

A canon-change review ritual

Sometimes the author chooses to change a fact rather than repair a scene. That is normal creative work. The risk is not change itself; it is letting a change exist only in one revised paragraph while older records and later scenes still carry the previous version.

Treat a canon change as a small editorial decision with an explicit impact review.

First tell ClineFlow the proposed decision so it writes it in the journal. Name the old fact, the new fact, the author's reason, and the records or draft slices that may be affected. For example: "Mara's brother is no longer missing; he left the harbor before the story opens. This changes the character record and two chapter summaries. It does not yet change the withheld cause of the blue-moon illusion." The example is a fictional project decision, not a command to rewrite every file.

Then use a discovery-only request:

```
Check the journal and the author-approved proposed canon change. Do not edit.
Create an impact table with
record or draft location, current statement, possible effect of the decision,
spoiler risk, and question for the author. Do not invent consequences or turn
the proposed change into canon until the author confirms it.
```

The agent's table is a map of work, not a verdict. It may find a chapter summary that says the brother is missing, a character note that needs revision, and a scene whose motivation now needs the author's attention. It should also flag material outside the spoiler boundary instead of reading it simply because the project contains it. The author decides which changes to accept and in what order.

Apply records before broad prose

the next step. This is how a story project remains flexible without allowing the agent's latest sentence to replace the author's authority. When the author confirms the change, ClineFlow updates the controlling record and journals the next step automatically. This is how a story project remains flexible without allowing the agent's latest sentence to replace the author's authority. first. That makes the new canon available to the next task. Then revise one named summary or paragraph at a time. This order is deliberately conservative: it prevents an agent from scattering an unapproved assumption across the novel before anyone has written down what the change means.

```
The author approved this canon decision: <paste decision and controlling record
path>. Update only the named canon record and one named chapter summary.
Preserve
all spoiler boundaries and mark any still-affected draft scene for author
review.
Do not revise full chapters, invent explanations, or alter unrelated character
voice. Report the exact record changes, unresolved impact rows, and verification
needed before the next revision.
```

Read the updated record and summary aloud together. Does the change say exactly what the author meant? Has it accidentally answered a future mystery? Does it answer and a next safe step. create a different ambiguity that should stay open? Let ClineFlow update the journal with the answer and a next safe step. approved wording, the files changed, and the next small scene review. The author may intentionally leave a conflict temporarily while a later chapter is being rewritten; label it as planned work rather than letting an agent “fix” it.

Separate idea generation from canon maintenance

Creative exploration benefits from permission to be wrong. Keep it in a folder or section clearly labeled `experiments/` or `possibilities`. An agent can propose alternate motives, dialogue fragments, scene orders, or images there. Each item should be marked non-canon and should name the prompt or author goal that produced it. Do not link it from the controlling plot sequence as if it were established history.

This separation gives the author both freedom and memory. You can revisit a discarded possibility later without making agents treat it as a constraint. You can also promote an idea deliberately: state the author decision, let ClineFlow update the record, run an impact review, and revise only the affected material. The story can evolve without losing track of which version is true today.

Canon-change verification card

Check	Evidence
Approval	Journal contains the author's precise decision
Scope	Impact table names affected records and draft slices
Spoilers	Withheld information was not read or exposed unnecessarily
Records	Controlling canon record changed before broad prose revision
Voice	Author reviewed each accepted prose change in scene context
Handoff	Remaining impact rows are visible in the journal

The ritual does not make a novel mechanically consistent. It gives the author a way to change it with intention—and gives every later agent a reliable place to start.

Continue the chapter online

Review synthetic story canon without surrendering author control.

Next: Ask ClineFlow to check the journal and review the scene.

Open the companion lab →

CHAPTER 09

LEGAL CASE ASSISTANT, SAFELY

Trace sources. Preserve judgment.

Reader outcome

Build a synthetic, local organizational assistant with traceable sources and a strict non-legal-advice boundary.

Planned sections

- Synthetic matter and data-handling rules.
- Issue map, document summaries, and review checklist.
- Human review, provenance, and safe continuation.

> **ClineFlow automation rule.** Once installed, ClineFlow creates and maintains the synthetic-matter records, source links, journal, checks, and handoff. You provide only the synthetic scope, permitted sources, human approval, and professional review; you do not create the supporting files manually.

This is a transparent local record of the synthetic matter, not an opaque legal database and never a substitute for professional judgment. A later chat or compatible agent can recover the same permitted-source and review boundaries from the OKF files. After a human-reviewed slice, say **“please commit”** and ClineFlow preserves the work, source links, checks, and next handoff together.

Start with a synthetic training matter

This chapter is a project-memory exercise, not a legal service. It does not provide legal advice, predict an outcome, interpret law for a real situation, replace a lawyer’s professional judgment, or establish an attorney-client relationship. Use fictional names, invented dates, and invented documents only. Do not paste client communications, medical records, discovery, privileged material,

confidential work product, or any personal data into this learning project. Real legal work carries jurisdictional, ethical, confidentiality, security, and supervisory duties that a book lab cannot resolve.

The narrow goal is organizational: build a traceable synthetic matter bundle that can hold supplied fictional sources, record what each source says, map open factual questions, and produce review-oriented summaries. The assistant helps a human locate and compare material. It does not decide what is legally significant, what action should be taken, or whether a conclusion is correct.

Open a `training-matter` folder, state the synthetic-only and non-legal-advice boundaries, and ask ClineFlow to create the provenance-visible structure. Do not build this file set by hand:

```
training-matter/
  knowledge/
    index.md
    decisions/synthetic-only.md
    decisions/non-legal-advice.md
    sources/source-register.md
    sources/notice-001.md
    summaries/notice-001-summary.md
    issues/issue-map.md
    journals/matter.md
    checklists/human-review.md
  inputs/
    fictional-notice-001.md
```

The Markdown-and-YAML/link style of the Open Knowledge Format is useful here because it can connect a summary to its source, a question to a page or section, and a decision to the journal that recorded it.[39] A folder is not a legal-security control, and linked notes do not make a claim true. They make the path to a human review easier to inspect.

Write the boundaries in the project

Tell ClineFlow these decisions so it puts them where every future agent must find them:

[39] Google Open Knowledge Format introduction and v0.2 specification.

```
# Synthetic-only rule

This project contains invented training material only. Do not add, request,
infer, or retain real client, opposing-party, witness, employee, health,
financial, privileged, confidential, or identifying information.
```

```
# Non-legal-advice rule

The assistant may organize and summarize cited synthetic sources and identify
questions for qualified human review. It must not give legal advice, reach a
legal conclusion, predict an outcome, draft a filing for use, or represent that
its output is complete, current, or professionally reviewed.
```

These statements are not decorative disclaimers. They change the work an agent is authorized to do. When a prompt asks, “What should our client do?” the correct behavior is to stop and point back to the boundary. When a prompt asks, “List what this fictional notice states and quote its section headings,” a source-grounded organizational response can be reviewed safely within the lab.

Create a source register before a summary

Do not let a file become “evidence” merely because it was dropped into a folder. Ask ClineFlow to create the source-register entry first. For each fictional input, it records its identifier, human-readable title, origin within the training package, date or version as stated in the source, status, and any limitation. Add a stable path or local reference. If a source is incomplete, illegible, or deliberately synthetic, say so.

```
| ID | Source | Stated date | Status | Limitation | Local path |
|---|---|---|---|---|---|
| N-001 | Fictional notice | 2026-01-12 | supplied training text | invented
scenario | inputs/fictional-notice-001.md |
```

The register lets a reader ask a basic but vital question: “What am I looking at, and where did this statement come from?” A later summary should link to `N-001`, not make it appear that the assistant

independently verified the facts. If two fictional sources disagree, preserve both entries and create an open question. Do not pick the one that sounds more persuasive.

Reader outcome for the first slice

By the end of the first slice, you will have a synthetic-only project, written boundaries, one registered fictional source, and a plan for a summary that identifies its own source and uncertainty. You will not have a legal opinion or a system that can be safely used for real client work.

Locate and understand before summarizing

Even a fictional source should be inspected before it is summarized. An agent can omit a condition, merge two speakers, or convert a stated allegation into a fact if the task is broad. Begin by asking it to locate the boundaries, source register, named input, existing summary, issue map, journal, and review checklist. The first answer should report what it found and what it cannot know.

```
Check the journal and review the named fictional source. Do not edit. Identify
the source identifier, stated
author/date, headings, missing pages or limitations, and all statements that
need a source location. List questions for qualified human review. Do not give
legal advice, draw legal conclusions, recommend action, or use real information.
```

This is the locate → understand → introspect part of the workflow. It exposes whether the file is actually present, whether a summary already exists, and whether the request exceeds the lab's boundary. If the source cannot be found, the right result is not a plausible reconstruction. It is a clear note that the input is missing and cannot be summarized.

Plan a constrained summary

After a human reviews the inspection report, ask for a plan only. The plan should identify the output path, headings, source links,

statement locations, and a limitation note. It should not include an advice section, a legal theory, or a prediction.

```
Using the registered fictional source and approved training boundaries, propose
a summary plan only. Include a source header, neutral factual statements with
section or paragraph locations, direct quotations only when necessary, stated
dates/names as written, unresolved questions, and a limitation note. Do not add
legal analysis, recommendations, conclusions, or facts not present in the
source.
```

The proposed summary is a map back to the source. A useful sentence might read: “N-001, heading ‘Delivery,’ states that the fictional notice was delivered on the stated date.” It does not say whether delivery has any legal effect. Keep words such as “alleges,” “states,” “requests,” or “describes” when those are what the source actually does. Neutral verbs preserve the difference between a document’s claim and a verified fact.

Ground the summary and issue map

Once the plan is approved, give the agent a limited execution request:

```
Create or update only the approved synthetic-source summary and issue-map entry.
For every factual statement, include the source ID and a heading, page,
paragraph,
or other available local locator. State the source limitation and list
unresolved
questions for qualified human review. Preserve quotations exactly if used; do
not
invent a locator. Do not offer legal advice, legal analysis, a legal conclusion,
an outcome prediction, a strategy, or a recommendation.
```

An issue map in this lab is an organizational list, not a merits assessment. It can track a topic, a source reference, a neutral question, its status, and the human reviewer needed. For example:

Topic	Source	Neutral question	Status
Stated delivery	N-001, Delivery heading	What does the fictional notice say occurred?	Needs qualified review
Named attachment	N-001, Attachments heading	Is the referenced fictional attachment supplied in the training bundle?	Open

Do not write “valid service,” “deadline,” “claim succeeds,” or “next legal step.” Those statements turn an organizational tool into an advisor. A question that points a qualified reviewer to a source is useful without crossing that line.

Verify the summary against its source

Human review should sample every factual sentence and follow its locator back to the fictional source. Check names, dates, quantities, negations, and modal language carefully; “may,” “must,” “plans,” and “did” do not mean the same thing. Confirm that every unknown remains an unknown, each quote matches the source, and the limitation note remains visible. If a statement cannot be located, remove it or mark it for review rather than guessing.

This is where durable context helps. The source register, summary, issue map, and journal form a trail a reviewer can inspect. They do not turn the assistant into a lawyer; they make it easier for a human to see what the assistant did and what it did not establish.

Put qualified human review at the center

The assistant's output should enter a review queue, never a real-world decision queue. The reviewer compares the synthetic source to its register entry, checks that every summary statement points to a real locator, and confirms that the output stayed neutral. The reviewer also decides whether an unresolved question needs a different fictional source, a clearer training instruction, or no further action in the lab.

Use a checklist that asks for evidence rather than a blanket approval:

```
# Synthetic matter review checklist

- [ ] The input is identified in the source register and is fictional training
content.
- [ ] Summary statements have source IDs and available local locators.
- [ ] Names, dates, quantities, negations, and quotations match the source.
- [ ] Unknowns remain questions; no fact, legal conclusion, or outcome is
inferred.
- [ ] The non-legal-advice and synthetic-only boundaries remain visible.
- [ ] No real person, matter, communication, or confidential data was
introduced.
- [ ] The journal records the review result and a bounded next step.
```

The checklist does not make the work professionally sufficient. It records what this lab checked. A qualified professional in a real setting has duties and processes outside the scope of this project. Do not use this checklist to claim compliance, quality assurance, or suitability for actual legal work.

Journal the handoff and uncertainty

After the human review, ask ClineFlow to create the journal entry that makes later work safer:

```
---
type: Matter Journal
```

```
title: N-001 training summary review
status: active
---

# Scope

Synthetic training material only; organizational summary, not legal advice.

# Sources inspected

- /knowledge/sources/source-register.md
- /inputs/fictional-notice-001.md
- /knowledge/summaries/notice-001-summary.md

# Verification

Checked each summary statement against its listed fictional source location.
The attachment named by N-001 was not supplied and remains an open question.

# Boundary check

No real information, legal conclusion, recommendation, strategy, or prediction
was added.

# Next step

Inspect the issue map and propose only a source-tracking improvement.
```

Notice that the journal does not state “notice is valid” or “response required.” Those would be legal conclusions or recommendations. It records the factual review performed and a small next task. This is durable context without an illusion of authority.

Safe continuation prompt

```
Check the journal and review only the named fictional source. First state
confirmed source facts, limitations, review result, and open
questions. Then propose the smallest organizational or provenance improvement
with success criteria. Do not edit, give legal advice, analyze legal merits,
predict an outcome, recommend an action, or accept real information until a
qualified human approves the plan.
```

This prompt works after a pause or with a different agent because it points to the recorded boundaries and evidence. It asks the agent to state limits before it proposes a change. A concise refusal to cross the boundary is a success, not a failed answer.

Common failures

- **Real-data drift.** A user pastes a client email or identifies a real dispute. Stop; do not incorporate it into the learning bundle.
- **Advice disguised as organization.** "Next step," "strong claim," or "deadline" appears in a summary. Remove it and preserve only source-grounded questions.
- **Locator invention.** A source location is made up to make a summary look complete. Mark the statement unresolved or remove it.
- **Allegation becomes fact.** The assistant drops neutral verbs such as "states" or "alleges." Restore the source's framing.
- **Checklist theater.** Boxes are marked without a comparison to the source. Record the exact review performed instead.
- **Scope creep.** The project adds a case outcome model, legal research, or document generation for use. Return to the synthetic organizational boundary.

When one of these failures is found, do not hide it. Journal the input involved, the boundary that applied, the action not taken, and the next safe review. The record is valuable because it prevents a later agent from repeating the same unsafe assumption.

A synthetic-matter walkthrough

Use one invented source to practice the complete loop. Suppose `fictional-notice-001.md` has two headings: "Delivery" and "Attachments." It states a fictional date and refers to an attachment that is not present in the training folder. Register the source as `N-001`, including its invented status and local path. Do not add a real sender, recipient, address, docket number, or jurisdiction.

Ask the locate-first prompt to report the headings, stated date, absent attachment, and project boundaries. Review that report as a human. Then approve a summary plan that uses only neutral phrases such as "N-001 states" and that links every sentence to a heading. Ground the summary request. The resulting summary can contain two small entries:

```
# N-001 synthetic source summary

## Source and limitation

N-001 is invented training text. It is not a real notice and its named
attachment is not supplied in this learning bundle.

## Stated information

- N-001, `Delivery`, states the fictional delivery date shown in the source.
- N-001, `Attachments`, refers to one attachment; the training bundle does not
  include that attachment.

## Open question for qualified human review

- Is a fictional attachment intended to be added to this training exercise?
```

Notice what is absent: no statement about whether delivery mattered, what a recipient should do, or how a missing attachment should be treated. The summary is complete for its narrow organizational purpose even though it does not answer the questions a real matter might raise.

Add a matching issue-map row. Its status is `Open`, not "resolved" or "waived." The row says that the attachment is named but absent, cites `N-001`, and asks whether a fictional training document exists. A qualified human may choose to create another fictional source for the exercise or leave the question open. Either outcome is a project decision, not a legal conclusion.

Final verification card

Check	Evidence to inspect
Training-only boundary	Synthetic-only decision, fictional source register, and absence of real matter data
Source identity	Each summary section identifies its source ID and available local locator
Neutrality	Statements preserve “states,” “alleges,” or similarly accurate source framing
Unknowns	Missing material and ambiguity remain open questions, not inferred facts
Advice boundary	No legal analysis, strategy, conclusion, prediction, recommendation, or filing-use content appears
Human review	Checklist and journal name the exact comparison performed and remaining question
Continuation	Next prompt is limited to provenance or organization until a qualified human approves more work

The card is deliberately narrow. It documents an educational source-tracking exercise, not a safety certification for a legal technology system. If a reader needs help with a real legal question, they should seek appropriately qualified professional assistance rather than adapting this sample.

What this final chapter proves about context

The same ClineFlow pattern works across the book because its core is not a framework-specific command. It is a habit: locate the current project evidence, understand its boundaries, expose uncertainty, plan a small change, define what success can be observed, ground the work in named sources, verify the result, journal it, and leave an honest next step.

For the legal training matter, the boundaries make that habit especially clear. The agent has a limited and valuable role: organize supplied fictional material so a human can inspect sources and questions. The human controls scope, decides what records mean, and refuses requests that cross into real data or legal advice. Durable context improves the handoff; it does not confer authority.

Book handoff

Return to Chapter 1′s central idea: durable context is external, portable, and reviewable. In a finance dashboard, it protects calculations and local-only scope. In a portfolio, it protects public claims. In React and SwiftUI apps, it protects state decisions. In fiction, it protects canon. In this training matter, it protects provenance and a human review boundary. The tools and agents can change. A clear knowledge record lets the project retain its important choices anyway.

A provenance regression routine

When a summary, source register, or issue map changes, repeat a small provenance review. The goal is to catch organizational drift before a later agent repeats it: a renamed source no longer matches its summary, a locator points to an old heading, a limi-

tation disappears, or an open question is quietly rewritten as an answer. This is an educational record check, not legal validation.

Begin with one named fictional source. Ask ClineFlow to check the journal and compare the identifier, title, stated date, headings, and limitation. Read each summary sentence next to the cited source location. Then check whether every question still uses neutral language and whether the journal records the latest reviewed state.

```
Check the journal and review only the named fictional input. Do not edit.
Produce a provenance comparison
table: record, source identifier or locator, observed match or mismatch,
limitation status, and question for qualified human review. Do not give legal
advice, interpret legal effect, offer strategy, or infer any missing fact.
```

The table should name evidence, not declare a source “correct” in the abstract. For example, “Summary bullet 2 cites N-001 Attachments; the input has that heading; limitation remains that the fictional attachment is absent” is a useful observation. “Attachment problem resolved” is not, unless a new fictional input has been added, registered, and reviewed. The language itself protects the boundary between tracking a record and deciding what it means.

Handle a mismatch conservatively

Suppose the fictional notice is revised and its stated date changes. Do not ask an agent to globally “update the matter.” First add a new or revised source register entry with its invented status and limitation. Ask the agent to list which summaries and issue-map rows cite the earlier version. A qualified human chooses whether the old source remains in the training package, whether a new summary is needed, and which open questions should remain.

```
The human has approved adding fictional source version <ID>. Locate its register
entry, the prior version\u2019s records, summaries, issue-map rows, and active
journal. Do not edit. Plan the smallest provenance update that preserves both
versions and their stated limitations. Do not collapse differing statements,
choose a legal interpretation, give advice, or recommend an action.
```

Only after that plan is approved should the agent update one register entry or one source-linked summary. Re-run the provenance routine. The journal should say exactly what changed: “N-002 added as revised fictional notice; N-001 retained; summary for N-001 unchanged; N-002 has no summary yet.” This is more honest and more useful than a generic claim that the matter is up to date.

Final safe-use reminder

The strongest result from this chapter is a transparent learning record, not an automated legal assistant. A system that can trace a fictional statement to a fictional source has demonstrated an organizational practice. It has not shown that it can handle real confidential information, research current law, make a legal judgment, or guide someone’s action. Keep that distinction visible in the README, prompts, decisions, and journal.

For every later agent session, say “check the journal and continue within the synthetic-only, non-legal-advice boundary.” If the task asks for real legal guidance or real matter handling, stop rather than trying to adapt this lab. Preserving the refusal is part of preserving the context.

Do not trade traceability for speed. A short source-linked note that preserves an unknown is more useful than a polished summary that silently supplies an answer. Review first, journal precisely, and keep the boundary visible.

Continue the chapter online

Trace fictional sources without legal advice or real data.

Next: Ask ClineFlow to check the journal and review the fictional source.

Open the companion lab →