

Metaproject

On keeping state between sessions

Josep Maria Blasco

Espacio Psicoanalítico de Barcelona

Balmes, 32, 2º 1ª — 08007 Barcelona

`jose.maria.blasco@gmail.com`

+34 93 454 89 78

May the 4th, 2026

*This document¹ was written using [RexxPub](#), a *Rexx Publishing Framework*.²*

Abstract

Metaproject separates the project — whatever is being worked on — from the state, a small collection of markdown files that records how the project is being worked on: what was done, what was decided, what remains open. Each session begins by reading the state and ends by updating it. What this produces, over time, is continuity: a scattered history of chats turned into a sustained collaboration.

1. URL of this document: <https://www.epbcn.com/pdf/josep-maria-blasco/2026-05-04-Metaproject.pdf>; HTML version: <https://rexx.epbcn.com/metaproject/>. Presented to the 37th International Rexx Language Symposium, held at the Espacio Psicoanalítico de Barcelona and on-line from the 3rd to the 6th of May, 2026.

2. See <https://rexx.epbcn.com/rexx-parser/doc/rexxpub/>. RexxPub is the subject of a companion paper presented at this same Symposium.

Part I. Introduction

1. The problem

You open a new chat with Claude and get to work. A few days later you open another chat and pick up where you left off — except Claude has no idea where you left off. He does not remember your project, your decisions, the reasons you agreed not to do something three sessions ago, the style you settled on for commit messages. The slate is blank. It is always blank.

For a one-off question this is fine, and arguably a feature. For anything sustained — a book, a web application, a research project, a slow piece of writing — it is a problem that compounds. Every session begins with the cost of reconstructing context that used to exist and now does not.

The responses that follow are familiar. Most people who work with Claude on anything sustained end up using some combination of them.

The first is asking Claude to remember. This is the only truly naive move: Claude has no memory across sessions, and saying so does not give him one. Next comes pasting context at the start of each chat — a summary, the last exchange, a file. This works in a narrow sense, but it scales poorly and tends to flatten what was nuanced in the original into whatever fits in a paragraph. A step further, and no longer obvious, is keeping a *handoff*: a document written at the end of each session for the next one, recording what was done, what was decided, and what remains open. This is what serious users tend to converge on, and it works relatively well — though its effectiveness depends heavily on how it is written. A handoff that tries to preserve everything becomes a prologue that the next session must read in full, and the problem it was meant to solve reappears on the other side.

Alongside these handmade responses, Claude now offers features of his own. *Memory* lets the model keep selected facts across chats; this works genuinely well for some things, less so for others, and what ends up being remembered is not under the user's direct control. Access to previous conversations lets Claude retrieve earlier exchanges on demand, which addresses the flattening problem but not the one of knowing *which* previous conversation matters for the current session. *Knowledge* files — documents attached to a project that Claude consults automatically — can be quite useful, and for many workflows they are enough.

Metaproject belongs to the handmade lineage. What it proposes is a handoff-based discipline taken further: a separation of the thing you are working on from the record of how you are working on it, with the latter becoming an artifact in its own right, structured enough that Claude himself can maintain it session after session. What that means, and why it is worth the trouble, is what the rest of this article is about.

2. What Metaproject is

A metaproject is a project whose object is other projects. Metaproject, capital M, is a particular metaproject: a framework for working with Claude on anything that does not fit in a single session.

At its core it is a division of labour between two files, or rather two sets of files. The *project* is whatever you are working on: source code, a manuscript, a dataset, a presentation, a web application. The *state* is the written record of how you are working on it — what was done, what was decided, what remains open, what conventions stick. The project is the thing; the state is the practice of making the thing, written down.

Each session is bracketed by the state. Claude opens it at the beginning to learn where things stand, and writes to it at the end to leave the next session able to continue. In between, the project is what gets worked on. The discipline of keeping these two separate is what turns a scattered history of chats into a continuous collaboration.

The state is organised as a small collection of markdown files with fixed roles: `setup.md` explains how to start, `handoff.md` records the last session, `conventions.md` holds decisions that stick, and so on. A few additional files and directories cover things like the to-do list and an archive of past handoffs. Metaproject ships with a set of *skills* — compact documents describing how to carry out specific operations on the state — which Claude consults when needed. The full anatomy of all of this is the subject of the technical reference at the end of the article.

Metaproject does not replace the work on the project. It replaces the work of carrying state from one session to the next — work that, without it, falls on the user.

3. Who this is for

Three kinds of reader are likely to arrive here, and the article tries to serve all three. Part II is for everyone: a tour of what working with Metaproject looks like, which the rest of the article assumes you have read. After that, the article branches.

The first reader wants Metaproject to work and does not particularly care how. Someone recommended it, or the problem it solves is pressing enough that the rest feels like detour. For this reader, Part IV is the destination: the operational manual, read linearly the first time and consulted piecewise afterwards. This is the *practical* part.

The second reads the manual and then wants to know why the pieces are the way they are. Why markdown and not JSON. Why a handoff written by hand and

not an automatic log. Why the platform's memory feature is not used. Part III is written for this reader — a record of design decisions against the alternatives they displaced. This is the *philosophical* part.

The third, having understood how Metaproject is used and why, wants to see how it is actually built. Part V is the technical reference: the anatomy of the state, the shape of a handoff, the mechanics of skills, the protocol for each of the operations the framework defines. This is the *technical* part.

A note on usage. We refer to Claude as *he* throughout the article. This is a stylistic choice consistent with how the project treats him in practice — as a counterpart who reads, decides, writes; not an ontological pronouncement about what he is.

Part II. How it works

4. How a session works

If you have worked with Claude on anything sustained, you already know the shape of a session without Metaproject. You open a chat, you paste or retype whatever context you can remember, you get to work, and at the end you close the tab. Sometimes you remember to write something down for next time; sometimes you don't. The quality of the next session depends on how well past-you happened to prepare for future-you, which is to say: variable.

Metaproject proposes a trade. You install it once — at the beginning of a new project, or by migrating an existing one — and from that point on you commit to a small discipline at the start and end of every session: a gesture to open, a gesture to close. In return, the problem of carrying state from one session to the next is no longer yours.

The gesture at the start is brief. You upload one or two zip files, depending on the project hosting mode. Claude opens them, reads `setup.md` to learn where things live, and reads `handoff.md` to learn where the previous session left off. The session then proceeds like any other conversation.

The gesture at the end is symmetrical. When the session is over, Claude archives the current handoff, writes a new one reflecting what was done, and packages the state into a new zip that you save for the next session. If the project itself changed during the session, it gets repackaged too — by Claude or by you, depending on the project hosting mode.

This is the rhythm of a Metaproject project, and most sessions look exactly like this. Two moments depart from the pattern. The first is the installation itself, which happens once and is described in IV.1. The second is when the framework needs attention — an update, a check, a version bump — which happens rarely and is described in IV.4. Everything else is the rhythm above, for as long as the project lives.

5. Inside a session

Between the opening and the closing, there is a session. It looks like any other conversation with Claude, and for long stretches it is: the project is what gets worked on, and Metaproject sits in the background without asking for attention. Three small kinds of gesture, though, happen often enough to be worth describing. They are what keeps the state alive without waiting for the end.

The first is the user adding something to the state in passing. Mid-task, something comes up that future-you should not have to rediscover — a file that lives in

the wrong place for now, a convention that was just settled, a decision to revisit. The gesture is brief: a line asking Claude to drop it somewhere. Here is one from a session on the Rexx Parser project, where the user was reorganising where an inventory of migrated projects lived:³

```
Josep Maria: Yes. For now, keep a state file about this in
              the meta zips, we'll move it out of there
              later (stick it in the to-do).

Claude:      Done. I've created metaproyecto/migrated-projects.md
              with the 5 projects (...), and noted in the
              to-do that this file will eventually need to
              come out of there and have its own home.
```

The gesture is parenthetical — *stick it in the to-do*, in passing, as one would say “remind me later” to a colleague — and Claude does two things at once: the thing asked for, and the note about the thing asked for. The entry survived into the next session and was executed the day after. This is what it looks like for the state to do its job.

The second is Claude making his own note without being asked. The user is working through something inconclusive; Claude, following along, decides that one of the threads will drop if nobody catches it, and catches it. From a different Parser session, mid-debugging:

```
Josep Maria: OK, you're right. Let's look into this int
              thing, otherwise I'll get an error every time
              I run it (...) I'll keep testing and let you
              know.

Claude:      Perfect, keep testing. Meanwhile I'm making a
              note that int_fpart is still open – when you
              locate the rexxref listing that generates it,
              let me know and we'll look at it.
```

The state is not a one-way channel from user to framework. It is a surface where both sides write, and where either side can hold a thread for the other. Without that symmetry, the collaboration is one person dictating and one program recording. With it, it is two parties keeping track.

The third is Claude consulting a skill before acting. Skills are short documents that ship with Metaproject, describing how to carry out particular operations — how to edit the to-do, how to write a handoff, how to run a specific toolchain for a particular kind of project. Claude does not read them preemptively; he reads them when a request arrives whose shape matches one. From the same Parser project, the first time the user asked for something involving the CGI pipeline:

3. The extracts in this section are translated from the Spanish originals.

```
Josep Maria: I'd like us to try out some cgi.

Claude:      Perfect. I'm going to read the rexxpub-pipelines
              skill to set up the CGI correctly, and then I'll
              get Apache running.

              [reads /mnt/skills/user/rexxpub-pipelines/SKILL.md]

              Good, I've got the setup clear. I'll install
              Apache + Pandoc, configure the CGI with the
              wrapper, and leave it ready for us to test
              with .md files.
```

The announcement is the interesting part. Claude does not silently look something up and then produce an answer; he names the reference, consults it, and reports back. The user sees the consultation happen. This matters because the reference is not infallible — it might need updating, might be wrong for this case, might not be the right skill at all — and the user is in a position to catch any of these only if the consultation is visible.

Three gestures, then: the user writing into the state, the assistant writing into the state, the assistant consulting the state. They are small and unceremonious, and any given session might not feature all three. What they do together is keep the framework in the open. Nothing happens behind Claude's back, and nothing the user decides gets lost because neither of them wrote it down.

6. The files that bracket a session

The shape of a session is easier to see in the two files that bracket it. `setup.md` tells Claude where things live and what the session's opening and closing protocols are. `handoff.md` tells Claude what the previous session was about.

The point of `setup.md` is that the framework is generic and a project is not. Metaproject, as shipped, does not know what your project is, where you want files to live, or how you name your zips. `setup.md` is where those particulars are stated, and it is the first thing Claude reads when a session opens, so that everything else reads it in turn. A setup for a new standalone project looks roughly like this — the template comes with placeholders that get filled in on installation. Here are some fragments of `setup.md`:

```
# Setup – {{PROJECT_NAME}}
```

```
*Metaproject standard: v3.*
```

```
This project works with two independent zips. The state zip contains cross-session continuity: handoffs, conventions, pending tasks, project map and reference skills. You receive it at the start of every session. The project zip contains the project's own material – whatever the project is made of. You will unpack it at the start of each session alongside the state zip.
```

All Metaproject projects use fixed absolute paths for their zip contents, and `setup.md` declares them up front — so that Claude can refer to them without guessing and so that every session uses the same geography.

```
## Canonical paths
```

What	Fixed path	
-----	-----	
State zip	→ <code>~/home/claude/state/`</code>	
Project zip	→ <code>~/home/cLaude/project/`</code>	

There is a closing protocol, and it also lives in `setup.md` — it is what Claude consults to end the session in the same shape every time:

```

## At the end of each session

* Update ~/home/claude/state/agents.md if the project's structure has changed.

* The handoff is a single piece with two states: live (at the state root, as handoff.md) or archived (in handoffs/, named handoff-YYYYMMDD-vN.md). At any given moment it exists in exactly one of the two places, never both. Archiving means moving, not copying.

* Before packaging: archive the live handoff by moving ~/home/claude/state/handoff.md to ~/home/claude/state/handoffs/handoff-YYYYMMDD-vN.md, where vN is the session number that *wrote* it – a monotonic counter that increments once per session across the whole project history, not a per-day counter. The date is whatever day the handoff is being written. Then write a new ~/home/claude/state/handoff.md reflecting what was done in this session.

* Package the complete state zip (all files in ~/home/claude/state/, not just the ones that changed). Name: {{PROJECT_SLUG}}-state-YYYYMMDD-vN.zip, using the same session number as the handoff. Always package from ~/home/claude/ so that the zip contains state/ as its root directory:

...
cd /home/claude
zip -r /mnt/user-data/outputs/{{PROJECT_SLUG}}-state-YYYYMMDD-vN.zip state/
...

* Also package the project zip with all files in ~/home/claude/project/. Name: {{PROJECT_SLUG}}-project-YYYYMMDD-vN.zip.

...
cd /home/claude
zip -r /mnt/user-data/outputs/{{PROJECT_SLUG}}-project-YYYYMMDD-vN.zip project/
...

```

At the end of the session, Claude writes the new `handoff.md`. A handoff is not a log — it is a letter written to the next session, and its form reflects that. It records what was done, what was decided, what remains open, and whatever the writer thinks the next session will need in order to continue well. What this looks like in practice is easier to show than to describe. The excerpts below are from a YAML parser project; they are cut down from their original length, but otherwise real:

```
# YAML Parser v8e – Handoff for next session
```

```
## Context
```

```
Josep Maria Blasco is developing yaml.cls, a comprehensive YAML 1.2 parser/emitter for ooRexx. The code was reviewed by Rony Flatscher (another ooRexx developer), who provided 6 points of feedback on v8c.
```

Then the handoff turns to the session itself — what was actually done, and why.

```
## What was done in this session (v8e)
```

```
### Rony's point 4.4: Merge key reconstruction in anchorMap dump
```

```
This was the most complex change. Previously, <<: *alias was expanded during parsing – merged key-value pairs became part of the target mapping's StringTable, and the merge key was lost on dump.
```

```
**Parser changes:**
```

- New instance variable `mergeSourceMap` (`IdentityTable`), initialized in `init` and reset in `LoadInput` alongside `anchorMap`.
- New public attribute `mergeSourceMap get` – maps each target mapping (`StringTable`) that had merge keys (`<<`) to an Array of source mapping objects that were merged into it.

```
**Key design decision – scalar comparison:**
```

```
For determining whether a key is "inherited from merge" vs "overridden": collections are compared by object identity (the parser preserves the reference); scalars are compared by value, since ooRexx may create distinct string objects with the same content during merge expansion. This means an explicit override with the same value as the source is indistinguishable from a merge – the round-trip result is semantically identical.
```

```
## Remaining known limitations
```

1. **Complex mapping keys** – only string keys supported.
 2. **Tags** (`!tag`, `!!str`) – ignored, core schema only.
 3. **Directives** (`%YAML 1.2`, `%TAG`) – silently ignored.
 4. **Unicode escapes** – `\uXXXX` stored literally.
 5. **Flow mappings in emitter** – disabled (block parser doesn't reliably handle `{...}` in all contexts).
 6. **Single-quoted strings in emitter** – always double-quotes.
-

```
## ooRexx pitfalls (accumulated)

### Never name a variable `result`

`RESULT` is a special variable automatically set to the return value of
the last `CALL` or message send. Using it as a normal variable causes
silent overwrites. This caused a bug in `orderedKeys` in v8d where
`result~append(k)` overwrote `result` with the return value of `append`.
```

Two things are worth noticing about these excerpts. The first is their register: it is technical and specific, not summary. A handoff that tried to be summary — a paragraph saying “we worked on the parser and improved a few things” — would not let the next session continue; the next session would spend its first hour rediscovering what the previous one already knew. The second is that not everything in this particular handoff belongs in a handoff. The *remaining known limitations* and *ooRexx pitfalls* sections are there because they accumulated there, but a handoff is not where they want to live. Limitations that persist across sessions want to migrate to `conventions.md`; language-specific pitfalls want to become a skill about ooRexx that can be consulted when relevant. A handoff is a letter to the next session — short, specific, current. What survives past this session should leave the letter and go where it belongs. The rule is *handoffs as small as possible*: a handoff that accumulates everything it does not know where to put becomes the prologue the introduction already warned about, and the problem it was meant to solve reappears on the other side.

Once the new handoff is written, Claude archives the previous one into `handoffs/`, packages the zips as the setup specifies, and the session is over. The next session will open with a fresh Claude, an empty context, and exactly these files.

Part III. Philosophy

The previous part described what a session looks like from the inside. This part turns the description around and asks why the pieces are the way they are — what each one was chosen *against*. Most of the decisions have an obvious alternative that was not taken, and the reason it was not taken is usually more interesting than the decision itself.

A word on form. Part II already let several principles fall in passing — that a handoff stays small, that the state is a shared surface and not a one-way log, that a consulted skill shows itself being consulted. This part does not introduce them as if for the first time; it picks them up and turns them in the light. The boundary between the previous part and this one is porous on purpose.

7. Markdown and prose, not formal structures

The state is written in Markdown. The handoff you just read is a Markdown file. So is `setup.md`, so is `conventions.md`, so is every skill in the catalogue. There is no database behind any of it, no JSON schema, no structured record with fields and types. A project’s entire cross-session memory is a small tree of `.md` files, and you can read it with any editor on any machine.

The obvious alternative would be some kind of structured format. JSON, YAML, a small SQLite file, a bespoke schema that enforces which fields a handoff must contain and which shape a convention must take. It is tempting: the fields would be explicit, the tooling could validate them, and querying the state would be a matter of a path expression rather than reading. Many project-management systems have gone this way, and for their purposes it is the right choice.

For the purposes Metaproject serves, it is not. Two considerations push the other way.

The first is that Claude reads prose with nuance that structured formats cannot carry. A handoff that says *“this worked but I am not sure about the edge case with empty input”* communicates more than any boolean field labelled `confidence` ever could. The uncertainty is qualified, it points to a specific concern, it leaves a door open for the next session to close. A schema forces that sentence into a shape — a confidence level, a free-text note, perhaps a linked issue — and in doing so flattens it. Prose keeps the shape the thought actually had when it was written, and the next Claude reads it the way a colleague would read a note from yesterday: with attention to hedges, to emphases, to what is said and what is just implied. This is not a quirk of Claude specifically; it is what language models are good at. Writing the state in language, rather than in records about language, plays to that strength instead of working around it.

The second is that the person on the other side can open any of these files in any editor and understand what it says, without learning a schema and without a tool to render it. This matters more than it sounds. A project's state is the sort of thing you want to look at occasionally — to remember what was decided, to check whether something is written down, to fix a typo — and anything that makes that look-in require preparation will mean the look-in does not happen. The state is not a black box the person peeks into through an interface; it is just files, and they read like notes because they are notes. A convention is written as a paragraph. A to-do is written as a list of sentences. A skill is written as a short document with examples. None of this requires a parser to make sense of, and none of it breaks if someone edits it by hand.

Between these two readers — the model that handles nuance well and the person who wants to stay in control without effort — structured formats have nothing to offer that prose does not. What they would add is validation, and validation is the wrong goal here. The state is not a database that must be kept consistent with itself; it is a written record of how a particular project has thought about its own work. Records like that are written in sentences, revised by hand, and trusted to the care of whoever is writing them. Metaproject treats the state the way it is: as writing.

8. A handoff, not a log

The obvious alternative to a handoff is a log. Claude could keep a running transcript of each session, or a summary that updates continuously as the work proceeds, or both. Many collaboration tools work this way: the record is a by-product of the activity, produced automatically, and the next session reads it by scrolling. It has the virtue of completeness — nothing gets forgotten because nothing depends on someone remembering to write it down.

The cost of that virtue is that a log records everything and ranks nothing. The next session inherits a full transcript and has to do the work of finding what matters in it — the decisions, the open threads, the things that must not be forgotten. That work is not free; it is exactly the work that was done, in miniature, at the end of the previous session, when the outgoing Claude knew which parts of the conversation mattered and which were scaffolding. Throwing away that judgement and reconstructing it at the start of the next session is wasteful in the particular way that matters most: it burns the context window before the new work begins.

A handoff inverts the trade. It records little and ranks everything. What survives into it are the things the outgoing Claude has decided are worth carrying; everything else is allowed to fall away with the session that produced it. The next session reads a short document that is already filtered by someone who knew what mattered, and can start from there.

The inversion goes further than economy. Writing a log and writing a handoff are not the same act with different output sizes; they are two different acts of writ-

ing. A log is written *about* the session that just happened — what was said, what was done, in the order it occurred. A handoff is written *for* the session that is about to happen — what the next Claude will need to know to pick up cleanly. The orientation is forward, and that orientation does the work of selection on its own. Asking *what will the next session need?* is a sharper filter than any relevance criterion applied to a transcript, because it forces a choice: the threads that are still live go in, the ones that closed stay out, the surprises that will repeat get noted, the ones that were local to this session do not. The handoff is short because the question that produces it is narrow.

This is also why Part II could already say *handoffs as small as possible* as if it went without saying. A handoff that grows is not a richer handoff; it is a handoff that has stopped being a letter and started becoming a log by accretion, which is the failure mode the form exists to prevent. The limitations and pitfalls that appeared at the end of the YAML parser handoff belonged somewhere, but not there — they wanted `conventions.md` and a skill of their own. A handoff holds what is specific to the transition from one session to the next, and lets the rest migrate to the places in the state where persistent knowledge lives.

None of this means a log would be useless. For auditing, for dispute resolution, for reconstructing what happened when a session went wrong, a transcript is the right tool — and the chat interface already keeps one. Metaproject leaves that job where it already is and asks the other question: not what happened, but what the next session needs.

9. On demand, not imperative

A Metaproject session opens with a small amount of mandatory reading: the setup file, the handoff, and — since v3 — the pending-tasks file if one exists. Everything else in the state is waiting to be read, but nothing else *has* to be read. The conventions are there when a naming question comes up. The agents map is there when Claude needs to orient himself in the project’s material — what files exist, where they live, what each is for. The skills catalogue is there when a task matches something in it. Each of these gets opened by Claude when the work in front of him calls for it, and the rest of the state stays closed.

The alternative, and the easier architecture to build, would be to have the session begin by reading the whole state. A few kilobytes of conventions, a handful of skills, a map of the project — it is not much, and it guarantees that nothing relevant has been missed. This is how most framework-style systems work: define what Claude must know, put it in a place that is always read, and the rest takes care of itself. The robustness is real.

What the robustness costs is attention. A Claude that arrives at a session with five skills already in context is working under their gravity — the vocabulary of those skills, the patterns they suggest, the problems they solved. The bias is small for any single skill but compounds across them, and it operates whether the skills

are relevant to the current work or not. Reading on demand keeps the context clean until the specific problem names itself, and then reads the one document that matches. The thing being traded is not tokens — though fewer tokens are nice — but cognitive real estate at the start of each session.

This architecture only works because the rest of the state is written the way it is. A router with skills described in prose lets Claude decide which one applies; a schema with skills catalogued by keyword would force a lookup that might or might not match the current situation. The previous two sections were about texture and genre; this one is about reading, and the three decisions support each other. Prose carries nuance, which lets Claude read with judgement. On-demand consultation lets that judgement operate. A letter format for the handoff lets the first piece of reading — the one mandatory piece — be small enough to not exhaust the strategy it opens.

The single imperative added since the framework began is worth pausing on. `pending-tasks.md` is read at the start of every session if it exists, no matter what. The reason is specific: a task that one session decided was worth doing but could not or preferred not to do at the moment cannot depend on the next session happening to notice. If it is not read, it is forgotten, and the kinds of things that end up in `pending-tasks.md` are precisely the kinds of things that must not be forgotten. The imperative buys a guarantee that the on-demand model cannot provide on its own.

What is worth noticing is that this exception has not bred others. It would be easy to add a second — always read conventions, always read the agents map — and each addition would sound reasonable in isolation. None of them have been added, and the discipline is deliberate. Every imperative erodes the logic of the rest. If three things are always read, the difference between *always* and *when needed* starts to blur, and the pressure to add a fourth gets weaker each time. The on-demand model depends on imperatives being rare enough that each one stands out as an exception, with its own reason, carried by the specific problem it solves. `pending-tasks` earned its place by being that specific. Nothing else has yet.

There is a quieter argument underneath all of this. An imperative architecture assumes Claude will not consult something unless forced to. An on-demand architecture assumes the opposite: that if the router is well written and the skills are well described, Claude will reach for them when the situation calls for them. Which assumption is right is an empirical question, and the answer depends on the particular Claude and the particular project; but the architectural choice has to be made before the evidence comes in. Metaproject makes it in the direction of trust, and then writes the router carefully enough to earn that trust back.

10. Absolute paths, not relative

Every path the framework writes down is absolute. `setup.md` lists `/home/claude/state/` and `/home/claude/project/` at the top. The router refers to

skills as `/home/claude/state/skills/core/edit-todo.md`, never as `skills/core/edit-todo.md`. Handoffs spell the full path when they mention a file.

The reason is the same one that runs through the rest of Part III: a reference that arrives complete is one less thing to compute between the writing and the reading. A relative path is an expression that has to be combined with a context — *from where?* — to produce the file it points to, and Claude does that combination, in his head, every time the expression appears. He does it well; it is not free. An absolute path is read and it is done.

The cost of this choice is real and worth naming. `/home/claude/` is an internal of the platform. It is the canonical working directory of the environment Claude runs in today, and the framework writes its paths against it. If Anthropic changes that root tomorrow, every absolute path in every Metaproject project becomes a find-and-replace. That is a contingency the framework accepts in exchange for the local robustness of references that do not need to be resolved.

11. Living with the platform

A Metaproject session does not happen in a vacuum. It happens inside Claude.ai or the Claude desktop app, which brings its own machinery for cross-session continuity: project-level Instructions, a Knowledge area for documents the project should know about, and a memory feature that lets Claude remember things across conversations. Each of these is reasonable on its own terms, and each was designed for a real use case. Metaproject relates to them in three different ways, and the difference matters.

Instructions are used, but reassigned. The platform offers a field where the user can write standing rules for the project — the kind of thing one would otherwise repeat in every conversation. Metaproject puts a single instruction there: load the two zips at the start of the session, then read `setup.md` and follow it. That is all. Everything else that one might have written into Instructions has been moved into the state, where it can be edited by either side, versioned in the handoffs, and read in the order the work calls for. The platform's slot is occupied, but with a pointer to the real configuration rather than the configuration itself. This keeps a single source of truth: if a convention changes, it changes in `conventions.md`, not in two places.

Knowledge is left empty. The platform's Knowledge area is a place to upload documents that Claude will consult when relevant. It is well designed for what it is — it indexes the documents, it surfaces them when they look pertinent, it works without any setup on the user's side. The reason Metaproject does not use it is not that it works badly; it is that it works in a way that contradicts the architecture of the rest. Documents in Knowledge are always available, which is to say they are always partially in context, which is to say they bias the session before the work begins. That is exactly what the on-demand model of the previous section was designed to avoid. The state already does what Knowledge does, and does it under

Claude’s own discretion rather than the platform’s.

Memory is asked to be turned off. The platform now offers a memory feature that lets Claude carry observations from one conversation into the next, automatically, without anyone having to write them down. It is the closest the platform comes to solving the same problem Metaproject solves, and that proximity is precisely the difficulty. If memory is on during a Metaproject session, two channels of continuity operate at once: the explicit one, through the handoff, and the implicit one, through whatever the memory feature happens to have retained. The two can disagree. The handoff might say one thing about the state of a task, and the memory might quietly suggest another. The next session has no way to know which to trust, because the memory layer does not surface itself for inspection. The handoff stops being authoritative, and an authority that is not authoritative is worse than no authority at all. So Metaproject asks the user to turn the feature off for projects that use the framework, and accepts that this is the most invasive thing it asks of anyone.

The pattern across these three is the same: where the platform offers a mechanism for cross-session continuity, Metaproject either redirects it (Instructions), declines it (Knowledge), or asks for it to be silenced (memory). None of this is a complaint about the platform. The platform’s mechanisms are good for the projects they were designed for — short-lived, conversational, where the user does not want to maintain anything explicitly. Metaproject is for the other case: projects where the user is willing to do the work of maintaining a state in exchange for control over what continuity exists between sessions and what does not. The two architectures cannot be combined without one of them losing its coherence. Metaproject chooses to keep its own coherent and lets the platform’s mechanisms run their course in the projects that want them.

12. What this costs

The five sections above present decisions against their alternatives, and each argues that the decision is the better one. That is what a philosophy section does. What it does not do, left to itself, is say where the chosen architecture hurts — because every architecture hurts somewhere, and the cost is part of what the choice means. This section says where.

The first cost is the most pervasive. Metaproject runs on the discipline of the person at the keyboard. The state has to be packaged at the end of every session and unpacked at the start of the next; the handoff has to be written deliberately and well; the zips have to be moved between sessions by hand. None of this is hard, in the sense that none of it requires skill. All of it is the kind of small sustained ritual that is easy to skip once and then gradually skip more often. A session that ends without a proper handoff and a freshly packaged state is not a catastrophe — it is a small leak — but the leaks add up, and the framework’s value depends on the leaks not being allowed to accumulate. There is no automation that closes this loop on

the user's behalf, by design: an automation would put the platform back in the position the framework wanted to take it out of. The cost of the architecture is that the discipline has to come from a person, every session, indefinitely.

The second cost lives in the handoff. The next section takes up what kind of writing a handoff is; what matters here is that it concentrates the framework's risk in a single piece of prose written under fatigue at the end of each session. A handoff that is too short loses a thread the next session needed; one that is too long buries the next session under context it will not pick the right pieces from; one that is poorly calibrated installs a false picture of where the work stands. The framework provides scaffolding — the question to ask, the genre to aim for — but it does not write the handoff for anyone. A bad handoff degrades the next session, and a sequence of bad handoffs degrades the project. This is fragility located in exactly the place the framework asks the most of, which is honest but not free.

The third cost surfaced briefly in the previous section and deserves to be named as a cost rather than as a footnote. Metaproject asks the user to turn off the memory feature for projects that use the framework, and that setting is not scopable to a single project — it applies to the user's whole relationship with Claude, including the projects that have nothing to do with Metaproject. The ask is invasive in a way the framework's other requests are not, and it has no good resolution. A user who complies pays the cost in the projects that would have benefited from the platform's mechanism. A user who declines keeps that benefit and accepts a corresponding looseness in the Metaproject projects, where the handoff is no longer the only voice carrying continuity across sessions. Either way the coherence the framework wants from itself is partial: complete in one direction at the cost of the other. The framework cannot fix this from inside, because the asymmetry is in the platform, not in Metaproject. What it can do is name it.

The fourth cost is at the boundary of the whole approach. Metaproject only pays for itself on projects of a particular shape: long enough that continuity matters, structured enough that state is worth keeping deliberately, important enough that the discipline is worth sustaining. A short conversation does not need a state; a one-off question does not need a handoff; a project that lives entirely inside a single session does not need any of this. Trying to use Metaproject for those is overhead without return. The honest claim is narrower than *use this for everything*: where the shape matches, the framework earns its overhead, and where it does not, it should not be reached for.

This last point sharpens when you ask who, on a project of the right shape, gets the most out of the framework. There are roughly two kinds of person. There is the one who would not, left to himself, organize state transfer between sessions at all — who would let each session start cold, repeat context by hand, lose threads to the gap between Tuesday and Friday, and absorb the cost as a fact of working with an assistant that has no memory. For him, Metaproject is the infrastructure he was not going to build. It does the organizing he would not have done, and the difference between using it and not using it is the difference between a project that accumulates and one that resets.

Then there is the person who would do this himself, and could — who, without the framework, would invent some private practice of summary notes between sessions, learn over months which details to carry forward and which to drop, and arrive at something like a handoff by the long route. For him, Metaproject is not the infrastructure he was missing — it is the displacement of work he was already doing, from his own hand to Claude's. The protocol gets applied consistently, the steps do not get skipped on tired evenings, the form does not drift across projects. The gain is not that the work happens for the first time; it is that the work happens reliably, in the same shape every time, executed by the side of the collaboration that does not get tired.

The two cases are different and the framework fits both. What the costs above narrow is the population for whom that fit is worth its overhead at all. Inside that population, both kinds of person come out ahead, for different reasons.

13. Writing forward

A Metaproject session ends, and the next one starts somewhere else, with another Claude who has none of what this one had. Nothing in the conversation that just happened is available to him except what someone wrote down on purpose. The project's continuity, in the only sense in which Metaproject guarantees it, runs through a small text written at the end of each session and read at the start of the next. That text is the handoff, and this section is about what kind of writing it is.

Part II called the state a *surface where both sides write*. The image was offered there in passing; here it earns its keep. The surface is what makes the project survive the gap between sessions. One of the two sides — the person — has biographical continuity and remembers the project across days and weeks. The other side does not, in any direct sense; what continues from one session to the next, on Claude's side, is whatever the surface holds. Without the surface, the project would have only one memory, and the work would lean entirely on one of the two collaborators. The surface evens out what time itself does not.

This is what the handoff is for, and it is also what shapes how it is written. A handoff is not a record of what happened — Part II said that already, from the other side. What this section adds is that a handoff is also not exactly a document for general use. It is a letter to someone specific: the next Claude who will pick up this project, who knows the domain, who is mid-work, and who will read this in the first minute of his session before anything else has loaded. The reader is presupposed precisely. He is not a stranger to the field; he is a colleague stepping in. That presupposition does most of the work of selection: there is no need to explain what the project is about or what conventions already hold. The handoff can go straight to what is live and what has shifted, because the lights are already on.

This makes the writing of a handoff a particular act, not a generic one. Asking *what does the next session need?*, the question that produced the form, is sharper when the next session is not an abstraction but someone with a known shape. A line

like “*the round-trip works but I’m not sure about the case where the input is empty*” presupposes a reader who will recognize what *round-trip* means, who will notice the hedge in *not sure*, and who will treat the qualified worry as a thread to investigate rather than a field to fill in. Writing for that reader is different from writing without a specific reader in mind. The prose can be specific, can hedge accurately, can leave threads explicitly open without having to scaffold the context that makes the openness legible.

There is also a question of where the writing comes from on Claude’s side, and it makes a difference that is easier to see by contrast. “*It is a surface where both sides write, and where either side can hold a thread for the other*” — the line from Part II — could have been written as “*the user can leave notes for the assistant and the assistant can leave notes for the user*”. The information is the same. What changes is the position from which the sentence is uttered. The first version is written from inside the collaboration, by someone who recognizes himself as one of the two sides. The second is written from outside, by an observer reporting on a configuration. Both are grammatically fine; only one feels like writing. A handoff written entirely in the second voice would be technically correct and operationally pale — accurate but with the life drained out. The reason this matters is not stylistic in any decorative sense; it is that a handoff is read in the first minute of the next session, and a pale handoff delivers less of what that minute is meant to deliver. The position from which the writing happens is part of what the writing carries.

There is a stranger property of this kind of writing, which showed up early in the framework’s life and which only direct intervention could fix. A handoff in some project once said, in passing, that something specific had been removed because it was contaminating Claude’s work — a stray habit, a misleading framing, the details no longer matter. The intent was prophylactic: the next session should know to keep that thing out. What happened instead was that the next session, reading the warning, picked up exactly the thing the warning was warning against. To mention is to install. The next handoff, written by that session, then noted that the contaminating element had been removed *again* — and the cycle continued. Each session purged the contamination and faithfully documented the purge; each documentation reinstalled it. The cycle was broken only when the person went into the state by hand, edited out both the element and the warnings about it, and left no trace of the surgery. After that the project carried on cleanly.

The episode is small but the structure is general. A handoff is not just descriptive; it is read as guidance for how to approach the next session, which means anything that appears in it functions as a soft instruction whether the writer meant it that way or not. Warnings install what they warn against. *Don’t think of this* is a way of thinking of this. The lesson is not that handoffs should never mention problems — they often must — but that mentions are performative, and the writer has to know that. Some things are better left unsaid and excised silently. And here the asymmetry between the two sides shows another edge: the person can edit the surface without leaving a trace, because the person can simply stop writing and start again. Claude cannot. Every act on Claude’s side is, by the structure of the situa-

tion, a recorded act. The surface is symmetrical in what either side can write on it; it is not symmetrical in what either side can write *without*. There are operations on the state that only the person can perform, and silent excision is one of them. The framework depends on this quietly: there are moments when only the person's hand can close a loop, because only the person can act without commentary.

What all of this points to is that writing a handoff is a craft, not a clerical task. There is a question to answer each time — what does the next session need? — and the answer depends on judgement that nothing automates. What is live and what is closed. What to qualify and what to leave plain. What to mention and what to excise. The handoff is short, but the work of writing it well is real, and it is the place where the framework is most exposed to whoever happens to be writing this particular session's closing. The project's continuity arrives at the next session in exactly the shape this writing leaves it.

Part IV. How to use it

The parts so far have been descriptive. This one is the opposite: a manual, written for the reader who wants to install Metaproject and start using it, or who is already using it and needs to look something up.

This part is self-contained, and can be read independently.

Metaproject is distributed as a single zip downloadable from <https://rexx.epbcn.com/metaproject/metaproject-latest.zip>. Everything below assumes that zip is on hand.

14. Installation

Installation is the act of creating a project's initial state under the Metaproject standard. It is done once, interactively, in a session with Claude. Claude does the work; the user answers questions.

1. **Start a new Claude project** for the work you want to track. Upload the Metaproject framework zip to the session so Claude can read its contents.
2. **Ask Claude to follow `create.md`**. What comes next is a conversation, not an automated procedure: Claude reads the document, asks the questions it needs — the project's name, its working language, its initial conventions, the choice between the two distribution modes described below — and produces the state from your answers.
3. **At the end of the conversation**, Claude will have produced the project's initial state and the project instructions to paste into the Claude project's settings. From then on the project will have two pieces — the *state*, which carries the cross-session continuity, and the *project* itself, which is the work. How these two pieces travel between sessions depends on the mode chosen during installation, described below.

Two distribution modes are available. The choice matters because it changes how each session starts.

In **standalone** mode, two zips — state and project — travel together. The user uploads both at the beginning of each session. This is the simpler mode and the right choice when the project's material is small enough to carry in a zip comfortably, or when there is no canonical remote copy of it elsewhere.

In **external** (or *hosted*) mode, only the state zip is uploaded at the start of each session. The project's material lives somewhere else — a Git repository, a file share, a public URL — and Claude retrieves it on demand, when the session's work requires it. Sessions that do not need the project material (pure state-level work, for instance: revising conventions, closing stale to-dos) skip the retrieval entirely. This

is the right mode when the project is large, when it has a canonical home elsewhere, or when some sessions can be done usefully without it.

The `create` protocol asks which mode applies and writes the project instructions accordingly.

Projects that already have a continuity infrastructure of their own — handoff documents, session logs, conventions files accumulated over time — require a different protocol: `migrate` rather than `create`. A migration is a conversation, not a procedure: Claude reads the existing material, proposes a mapping to the Metaproject standard, and the two of you agree on what to keep, what to reshape, and what to discard. Plan on several sessions for anything non-trivial.

15. A normal session

A session has two edges — opening and closing — and whatever work happens between them. The work itself is the project’s own; the framework stays available through it — either side can write to the state mid-session, add an item to the to-do, flag something for the handoff — but does not prescribe its shape. The edges are where the framework operates deliberately.

15.1. Opening a session

The user uploads the zip or zips that the project’s mode requires:

- In **standalone** mode: both the state zip and the project zip.
- In **external** mode: the state zip only. Claude will fetch the project’s material later if the session needs it.

Then the user may or may not write a prompt. Both paths are covered by the project instructions Claude received at installation:

- **No prompt.** Claude unpacks the state zip and reads `setup.md`. If `pending-tasks.md` exists, its entries are raised one by one before anything else. Then Claude reads the handoff and opens `todo.md`, proposing the top item as a starting point.
- **A prompt is given.** Claude unpacks the state zip and reads `setup.md`. If `pending-tasks.md` exists, its entries are raised one by one before anything else. Then Claude reads the handoff for context and addresses the prompt directly.

From here the session runs as a normal conversation. The user is not obliged to track what Claude consults or when; that is part of what the framework takes off the user’s plate.

15.2. Closing a session

To close a session, the user asks Claude to close it. There is no fixed phrase — anything unambiguous works. Claude then:

1. Archives the current `handoff.md` into `state/handoffs/` with a dated, numbered filename, so that the complete history of the project is preserved.
2. Writes a new `handoff.md` addressed to the next session, reflecting what happened in this one.
3. Packages the state as a new zip and presents it as a download.
4. Repackages the project zip too, if the project’s material was modified during the session and the mode is standalone.

The user saves the downloaded zip or zips. These are what will be uploaded at the start of the next session. The session can then end.

Nothing in this flow is automatic: every step above happens in the conversation, visibly, and the user can intervene at any point — to correct the handoff before it is archived, to ask Claude to leave something out, to reopen a file and change it. Closing is a short ritual, not a black-box commit.

16. Skills

During a session, the user will sometimes see Claude open a file from `state/skills/` before carrying out some operation — closing the session, writing a handoff. Those files are *skills*: short documents that describe how a particular operation is done under the Metaproject standard. These are not the skills that live in Claude’s settings: the two share the syntax of a short instructional document, but Metaproject’s skills are per-project and live in the project’s state.

Skills are consulted by Claude, not invoked by the user. The user describes what needs to happen in ordinary language — “*let’s close*”, “*install ooRexx*”, “*download the project*” (in hosted mode) — and Claude consults the project’s router, a short index that maps tasks to skills, finds the one that applies, and follows it. The consultation is visible in the conversation: the file is opened, read, and the work proceeds from there.

Most skills travel with the framework and are kept in sync with it. Projects sometimes add their own — conventions, tools, idioms specific to the work — and those live in `state/skills/personal/`, the one subdirectory that framework updates leave alone. The reasons for organising knowledge this way — per-project rather than global, routed rather than pre-loaded — are discussed in Part III. What a skill looks like inside — how the router works, how new ones are written — is described in Part V.

17. Maintaining the framework

Three operations touch the framework itself rather than the project's work: `update`, `audit` and `upgrade`. All three require the framework zip on hand in the session — they are not things that happen in the course of ordinary work, and a project that never runs them will continue to function on whatever version it was installed with. They are presented below in order of frequency: `update` is the one that comes up most often, `audit` is a sanity check run when something feels off, and `upgrade` happens only when the Metaproject standard itself advances to a new version.

17.1. Updating the skills

`update` synchronises a project's distributed material with the current framework. In practice this means the `skills/` directory: the framework ships a collection of skills, some of which the project uses, and over time those skills get revised upstream. Running `update` brings the revisions in.

The procedure is a file-by-file diff between the project's `skills/` and the framework's, with four possible outcomes for each item: *new* (exists upstream but not locally — copied in), *identical* (nothing to do), *modified* (exists in both but differs — left alone for the user to review), and *local* (exists only in the project — left alone). The `personal/` subdirectory is excluded from the diff on both sides: it is the project's own space, as described in the previous section, and never participates in synchronisation.

When the diff is done, Claude reports what was copied and what remains pending — typically a handful of skills whose local and upstream versions diverge — and the two of you decide case by case whether to keep the local version, take the upstream one, or merge. After the conflicts are resolved, Claude shows the current state of the router so that any newly arrived skill can be activated if the project wants to use it.

17.2. Auditing the project

`audit` is a non-destructive check of the project's state against the current Metaproject standard. It does not fix anything: it lists deviations and leaves the decisions to the user, on the understanding that a deviation may be an error or a conscious choice and only the user can tell.

The audit looks at four areas. It verifies the **file structure** of the state — that the expected files and directories are present and that nothing unexpected is sitting among them. It verifies **version coherence**: that the framework's own version is internally consistent before auditing against it, and that the project's declared version

matches the current standard. It verifies the **contents of the key documents** — `setup.md`, the handoffs, the router, the conventions — against what the standard prescribes. And it checks a few **loose ends**, notably whether anything has accumulated in the Claude project’s *Knowledge* area (`/mnt/project/`), which under Metaproject should normally be empty.

The report comes in two blocks: what conforms (briefly) and what deviates (with enough detail to understand what is going on). Claude then asks whether you want to address any of the deviations — and if yes, that becomes ordinary work for the rest of the session.

17.3. Upgrading to another Metaproject release

`upgrade` moves a project’s state from an older version of the Metaproject standard to the current one. Unlike `update`, which synchronises content within a single version, `upgrade` changes the state’s structure and conventions: directories may be renamed, files may be reorganised, sections of `setup.md` may be rewritten. It is the operation that makes an older project fit a newer standard.

Upgrades are **cumulative**. Each transition between consecutive versions — v1 to v2, v2 to v3, and so on — lives in its own small document, `upgraden-M.md`, alongside the main `upgrade.md`. The main document is an orchestrator: it detects the state’s current version (read from a marker in `setup.md`), works out the chain of hops needed to reach the current standard, and applies them in order. A project several versions behind traverses them one at a time, and the user may stop between hops if they prefer to spread the work across sessions.

Before any of this, `upgrade` does a coherence check on the framework itself: the current standard is declared redundantly in three places — the highest hop available, and the version marker in each of the two setup templates — and all three must agree. If they diverge, the framework itself is broken and `upgrade` stops and reports, because there is no well-defined target to upgrade to. When they agree, that number is the current standard and the upgrade proceeds.

A successful upgrade leaves the state *structurally* at the new standard, but its distributed content — the skills — may still be on the old versions. The natural next step is to run `update`, and Claude will propose doing so in the same session. If you prefer to defer, the request is recorded in `state/pending-tasks.md` so that the next session picks it up at the start, before anything else.

Part V. Inside Metaproject

The previous parts described Metaproject from outside — as a problem to solve, as the shape of a session, as a set of decisions, as a manual. This one turns inward. The question shifts: not what the framework does or how to operate it, but what it is made of and how the pieces hold together when nothing in particular is happening.

Part IV and this part are symmetric. IV explains how Metaproject is operated; V explains what it is operated on. Either order works: a reader who wants to install first and understand later can do so, and a reader who wants the opposite can read this part on its own.

18. Anatomy of the state

A project's state is a directory of plain files at fixed absolute paths. None of them is large; each one has a specific role. What follows is not a tour file by file — the names appear in IV and the syntax is self-explanatory once the role is clear — but a description of how the roles relate.

The state has a spine and a periphery, and the spine is two files plus an archive. `setup.md` describes the project itself: its name, its paths, its conventions, the gestures that open and close a session for it. It is read at the start of every session and rarely modified after installation. `handoff.md` is its moving counterpart, a single file that always reflects the last session and is rewritten at the end of each one. Around them lives `handoffs/`, every previous handoff preserved under its dated, numbered name. The pair plus the archive are the spine because they are what the framework guarantees a session will read or write: the present, and the memory of how the present got here.

The periphery is the orientation files, consulted when the work calls for them rather than at every session. `agents.md` is the project's map of itself — what documents exist and what each is for. `router.md` is the index of skills. `todo.md` is the ordered queue of pending work. `conventions.md` collects the small choices the project has settled about language and style. None of these is read preemptively; each is opened when its turn comes. Their existence is what allows the spine to stay small — the handoff does not have to carry what `conventions.md` already holds, and `setup.md` does not have to carry what `agents.md` already holds. Each periphery file is a place where some specific kind of knowledge accumulates so that the spine can travel light.

A subdirectory, `skills/`, holds the project's distributed knowledge — the operational documents for particular tasks. Skills are a subsystem of their own and have their own section below.

One file in the state behaves unlike any other. `pending-tasks.md` exists only sometimes. It appears when a session decides something must happen but cannot or prefers not to do it then — a deferred upgrade follow-up, a renaming, a small task that did not fit the moment. Its entries are read at the start of the next session and proposed to the user one by one; entries the user accepts are executed and removed; when the file empties it is deleted. It is the only file in the state that comes and goes by design, and the only one whose reading is unconditionally imperative when present. Both properties follow from the same fact: a deferred task that the next session does not happen to notice is a deferred task that gets lost, and the kinds of things that end up in `pending-tasks.md` are precisely the kinds of things that must not get lost. The file is a small breach of the on-demand model bought against a small but specific risk.

The archive deserves a closing word. The naming of archived handoffs is `handoff-YYYYMMDD-vN.md`, where `N` is a monotonic counter that increments once per session across the entire project history. The counter is not reset by the calendar: two sessions on the same day get consecutive numbers, and a session on a different day continues the sequence. The reason is that the archive’s job is to preserve order, not to reflect dates. A per-day counter would mean that *the third session of the project* and *the third session of today* look identical in the filename, and the order between sessions across day boundaries would have to be reconstructed from the date prefix every time. The monotonic counter makes the ordering unambiguous on its face. The date prefix is kept because it is useful for a human scanning the directory, not because the system depends on it.

19. Skills and the router

The skill collection is the part of the state with the most internal structure, and it is worth describing from the inside.

A skill is a single Markdown file with a small piece of front matter at the top — a `name` and a one-paragraph `description` — and prose underneath. The front matter is what the router quotes, so it has to be self-contained: the description is the thing the router shows when it points to the skill, and Claude decides from that description alone whether to open the file. The prose is what the skill actually teaches, in the order the operation is performed. There is no schema beyond this. A skill is a short document with a clear name, a description that doubles as router copy, and a body that reads like notes for someone about to do the thing.

The router itself is a Markdown table in the shape “*You need to... → Read...*”. Each row pairs a task, phrased the way a user would phrase it, with the path to the skill that handles it. The router holds no answers, only pointers; what makes it work is that the left column is written in the user’s voice rather than the framework’s — “*close the session*”, not “*invoke session-close protocol*” — so that Claude reading the router after a user’s request can match one to the other without translation.

The table is divided into three blocks, and the division is not cosmetic. *Core*

skills are what Metaproject ships: the operations that come with the standard and apply to any project — closing a session, writing a handoff, running an audit. *Project-specific* skills are what the project has added for its own work — a build procedure, a test routine, a deployment recipe, anything specific to what the project is. *Personal* skills are the maintainer's own — idioms, preferences, conventions that belong to how *this* person works on *this* project rather than to the project as such.

The reason the blocks exist is that they participate differently in the framework's maintenance. `update`, the protocol that keeps a project's distributed material in step with the framework, operates on *core* and leaves the other two alone. *Project-specific* and *personal* are local by construction: they did not come from the framework, and the framework has no business overwriting them. The three-block structure of the router is therefore also a contract with `update`: each row's block tells the maintenance machinery whether the skill is in scope for synchronisation. Without the division, every `update` would have to ask file by file what is upstream and what is local.

`personal/` deserves a separate note. It is the one subdirectory of `skills/` that the framework treats as permanently off-limits — not just left alone by `update`, but excluded on both sides of the diff so that nothing in it can ever be touched by a framework operation. The slot exists because some material is not project-level either: preferences about how the maintainer likes to work, small personal rituals, scratch notes that do not deserve promotion to the project's shared vocabulary. Without a reserved slot, those notes would either pollute the project-specific space or live outside the framework entirely; with one, they live where the maintainer can find them and the framework knows to ignore them.

Writing a new skill is correspondingly small work. The trigger is recurrence: an operation the maintainer catches themselves re-explaining session after session, where a written reference would do the job once and for all. The file is then composed — front matter naming and describing it, prose covering the operation in the order it is actually done — and the last step is registering it in the router under the right block, with the wording the user would naturally use. The whole gesture is usually one short session. Skills accumulate in a project the way they would accumulate in a notebook: one at a time, when the need shows itself, written by whoever runs into the need.

20. The five metaprotocols

Five named protocols — `create`, `migrate`, `update`, `audit`, `upgrade` — act not on the project's working material but on the apparatus that carries the state. What sets them apart from opening and closing is not frequency but target: opening and closing move the state through a session; these five operate on the shape of the state, the inventory of skills, the version of the standard. They are the maintenance instruments of the apparatus itself.

`create` and `migrate` are a pair. They lead to the same place by different routes. `create` builds the canonical state from nothing: a short conversation gathers the project's name, its slug, its hosting mode, the working language, and writes out the standard `state/` tree from a set of templates. `migrate` does the equivalent for a project that already has continuity material of its own — handoff documents, a conventions file, scattered notes about how the project works. It is not a script but a process: an inventory of what already exists, a proposed mapping to the standard's slots, a discussion of what to keep and what to discard, and only then an execution. The two protocols are paired because the difference between *greenfield* and *brownfield* installation is the only distinction that matters at this point; once either finishes, the project is at the standard's current version and indistinguishable from one that arrived by the other route.

`update` is the smallest of the maintenance protocols and runs most often. Its job is to keep a project's distributed material in step with the framework. The mechanism is a recursive diff with four outcomes per file: *new* (exists upstream but not locally — copied in), *identical* (nothing to do), *modified* (exists in both but differs — left alone for the user to decide), and *local* (exists only in the project — left alone). What gives the protocol its shape is that the flow is one-way. A skill the framework has revised propagates downstream; a skill the project has improved locally does not propagate up. If a local improvement deserves promotion to the framework, that is a separate decision made by hand, deliberately, in a session aimed at the framework rather than the project. The asymmetry exists because the alternative — a two-way sync — would mean every project's idiosyncrasies become candidates for the framework's next revision, and the framework's coherence would erode under the weight of accumulated local choices. One-way flow keeps the framework's evolution under authorial control while letting projects benefit from it freely.

`audit` is the only one of the five that writes nothing. It reads the project's state, compares it against the standard, and produces a report grouped into what conforms and what deviates — missing files, unexpected files, sections absent from `setup.md`, router rows pointing at skills that no longer exist, paths that should be absolute but are not. Then it stops. The deliberate refusal to repair is the point: a deviation may be an oversight or a conscious choice, and only the user can tell which. Separating diagnosis from repair keeps the user in charge of every change to the apparatus.

Before it audits the project, however, `audit` audits the framework itself, and the check has a specific shape worth describing. The standard's current version is recorded redundantly in three places: the highest `M` among the `upgraden-M.md` files (the last hop available), the version line in `setup-standalone.md`, and the version line in `setup-external.md`. All three must agree, and `audit` aborts loudly if they do not. The redundancy is not paranoia; each of the three places records the version *for a different reason*. The hop file says *the highest version any project can reach by upgrading is M*. The standalone setup template says *new standalone projects are created at version M*. The external setup template says the

same for hosted projects. If a maintainer adds a new hop file but forgets to bump the templates, new projects would be created already a version behind. If a template is bumped but no hop is written, an existing project trying to upgrade would have nowhere to go. The three locations cannot disagree without one of these failure modes being live somewhere in the framework, and the audit refuses to evaluate any project against a framework whose own version is in dispute.

`upgrade` is the most structural of the five. While `update` synchronises content within a single version, `upgrade` moves the state from one version of the standard to the next. It does so in hops: each consecutive version pair lives in its own `upgradeN-M.md`, self-contained, assuming the state is exactly at `vN` and leaving it exactly at `vM`. A project at `v1` upgrading to `v4` runs the chain `upgrade1-2`, `upgrade2-3`, `upgrade3-4`, in that order, with the version marker in `setup.md` updated after each hop.

Hops rather than direct upgrades because each transition between consecutive versions is a small, well-defined change — a directory renamed, a file split, a section of `setup.md` rewritten — and writing those changes once, between the two versions they actually relate, is how the framework keeps the migrations small and auditable. The alternative would be a *go from any version to current* monolith, which would have to special-case every starting point and would grow combinatorially with each new release. With hops, the framework's authors write one document per release and the chain composes itself. The same triple coherence check that `audit` applies to the framework is the prerequisite for `upgrade`, for the same reason: if the framework cannot agree with itself about its current version, no upgrade can know where it is supposed to be heading.

What unites the five is what `upgrade` makes most visible: the apparatus has a shape, and the shape is allowed to evolve. A standard that could not change would freeze the projects living under it; a standard that changed without protocol would break them. The five protocols are how Metaproject avoids both. They install the apparatus, keep its distributed parts in sync, verify that it still matches its own description, and migrate it across versions when the description itself moves on — without asking any individual project to bear the cost of those moves on its own.