INSTANT RECALL: Finding What the Agent Already Owns

Oscar Serra · Independent Research

21 October 2026

Abstract

You mention that a download is blocked, and your assistant reaches for the browser helper you built together last month. Nobody had to tell it "you have a skill for this".

Assistants keep plenty and still miss things like that. What they miss is something you own, a skill, a machine, a decision or a way of solving a problem, tied to the request by what it is for and often sharing no word with it. A search tool helps only when the agent suspects it has something; context pushed at the head of every turn fills with old documents and breaks the prompt cache.

We propose a design for the finding stage of an agent's memory: an index of what you own, keyed by the situations each thing helps in, that pushes a few pointers into every turn and follows a "last time" back to the session it means. They go after the part of the prompt the provider has cached, and they include only what the person or agent receiving the answer is allowed to see.

For you this means fewer reminders, a correction your assistant can find on your very next request, and no extra cost from rewriting the cached part of the prompt. This is a design; Section 9 names the tests that would decide it, on a replay of your own corrections, against the best design we know, one that searches every asset's own description.

If it holds, your assistant stops asking what you mean by "the one we did last time".

1. Introduction

Three requests arrive on one morning.

"Download last quarter's report from the company portal." The agent answers that it cannot reach the portal. A month earlier the owner and the agent built a helper that drives the owner's own browser session for exactly this kind of site. Its notes talk about "relaying commands to a logged-in browser" and never say "download".

"Check the camera on the lab machine." The agent says it has no credentials. It has them: a note about that machine says where its login is kept.

"Train it again with the setup we used last time." The agent asks which setup. The answer sits in a session record from three weeks ago, filed under another project's name.

Each time the owner has to say it: you have this. We call it an **ownership miss**: the agent has what the task needs and does not reach for it. Each one costs the owner his attention and teaches him to do the remembering himself.

The three fail in different places. The first shares no word with its request: what links "download the report" to "relay commands to a logged-in browser" is a use, *when a site blocks a download*. The second names the machine, but nothing looked the name up. The third names nothing at all: "last time" points at a session, not a topic. Retrieving the text nearest to a request finds what the request is about; these misses need what a stored thing is *for*, a name that is always checked, and a memory of the task in progress. A search tool helps only when the agent suspects it has something, and context pushed in front of the model is paid for on every later call and breaks the provider's cache if it lands inside the cached prefix.

We propose a design for the finding stage, Find for short (Figure 1), with four claims, each decided by a test in Section 9.2. A miss can be placed at the first of six gates it fails (§3). An index of what the owner owns, keyed by use as well as topic, finds what topic search misses (§4, §5). A short list of eligible pointers pushed after the cached prefix costs a small fraction of the searches it saves and never rewrites that prefix (§6 to §8.1). And a correction becomes findable on the next request, before any index is rebuilt, and stays findable through an index swap (§8.2, §8.3). Table 1 fixes a few words, and Table 2 shows what this would change on that morning.

Table 1. Words used here.

Word	Meaning here
owner	whom the agent works for; he owns the assets, and the agent has them to use
reader	the session that will see an answer: the owner's chat, a sub-agent, a shared chat
asset	a skill, tool, machine, decision, past approach or past session the owner owns
pointer	a line naming an item and where it lives, without its body
cached prefix	the start of the prompt the provider keeps and bills cheaply; the turn's part follows
use key	a short phrase saying when an item helps, such as *when a site blocks a download*
named slot	places at the top of the pushed list, kept for items the owner named
task-reference state	what a session keeps about its task, so that "last time" can be followed back
session card	a note on a finished session: what it did, the files it touched, its dates
fresh-writes overlay	the writes since the last index build, checked on every request
eligible	allowed for this reader: in scope, not quarantined or suppressed, valid at the date
index swap	replacing the live index with the one built overnight

Table 2. The morning of §1, today and with Find.

Request	Today	With Find	Caught by	Decided by (§9.2)
"Download last quarter's report from the company portal."	"I can't reach the portal."	A pointer to the browser helper, in front of the model before its first call	The use key *when a site blocks a download* (§5)	Use keys against none
"Check the camera on the lab machine."	"I have no credentials."	The machine's entry, which says where its login is kept, never the login	Its name, in the named slot (§6.4)	The named slot, every case
"Train it again with the setup we used last time."	"Which setup?"	The card of the last session on that task, pointing at the setup	The task-reference state, which sends "last time" to the named slot (§6.1), and session cards (§6.9)	"Last time" resolved right; asking a session against searching
"You have a skill for this."	The owner says it again next week	The request before the correction becomes a use key, findable on the next request	Keys cut from corrections (§5) and the fresh-writes overlay (§8.2)	A correction retrievable at once, and pushed on a repeat, every case

The hippocampus gives us a frame, a way to reason and not a mechanism we copy. On the indexing theory (Teyler and DiScenna, 1986) it stores an index of where the parts of an experience live, and a partial cue brings back the whole pattern (Marr, 1971). A cue works best when it matches how the memory was encoded, the *encoding specificity* principle (Tulving and Thomson, 1973): a name comes back more easily in the room where you learned it. The design takes all three: an index kept apart from the store, a pointer that brings back the whole item, and cues stored with each memory for the situations that will need it. The title's Instant Recall is the aim: the right thing in front of the agent before anyone asks.

2. Related work

This is one of four designs for an agent's memory: keeping (J1, the session record and what survives the context window), writing (J14, what enters long-term memory), finding (J2) and learning (J5, turning experience into behaviour). They share one transition specification (Appendix A). Nothing here has run on real memory.

Memory the agent searches for itself. MemGPT (Packer et al., 2023), now Letta, keeps a few memory blocks in the window on every call and leaves the rest to search (Letta, 2026). Claude Code loads the first 200 lines of its memory index into every conversation and reads topic files on demand (Anthropic, 2026a); Claude Managed Agents mount each memory store as a directory and describe it in the system prompt (Anthropic, 2026e). Anthropic's Tool Search Tool keeps deferred tools out of the prompt until the model searches their descriptions, which raised accuracy on Anthropic's MCP evaluations with large tool libraries from 49% to 74% for Claude Opus 4 (Anthropic, 2025b). Each pushes an index and leaves the body to the agent, the shape our cost model arrives at (§7), and each waits for the agent to search: a search it never thinks of is gate 3 (§3).

Keys that say when. Agent Skills load every skill's name and description at startup (Anthropic, 2025a), and Anthropic's authoring guide asks each description to say when to use

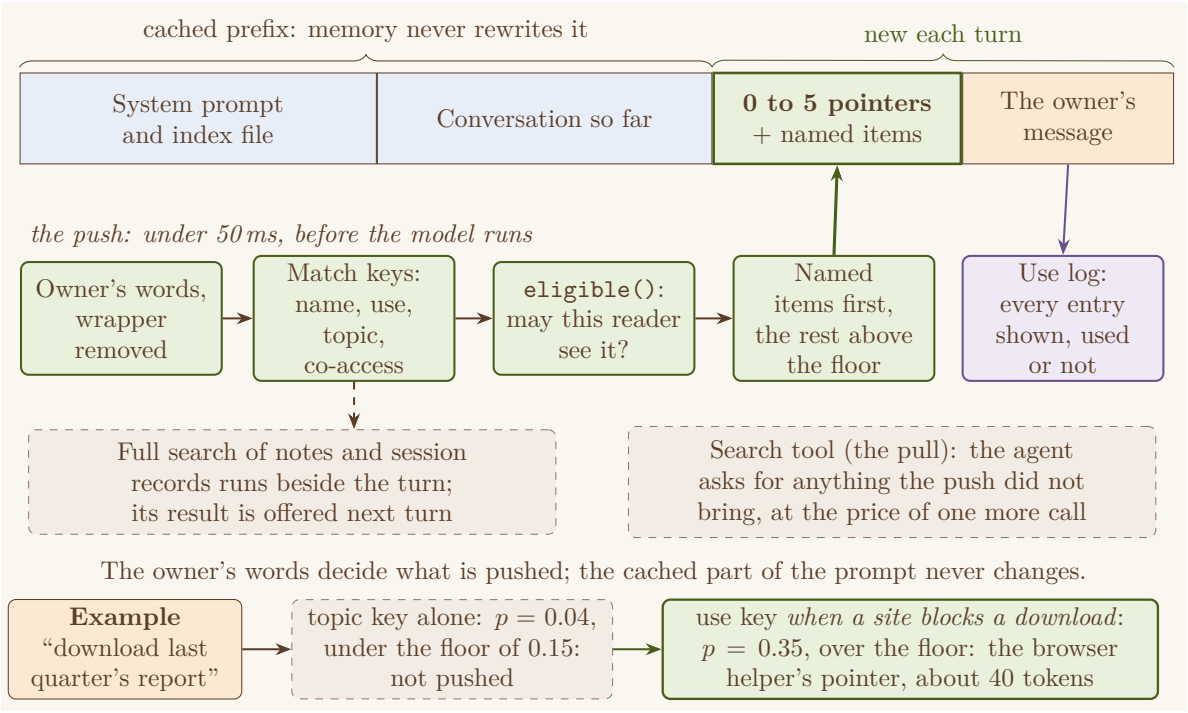


Figure 1. One turn of Find, drawn as the prompt the model receives. The system prompt, the index file and the conversation so far form the cached prefix, which memory never touches. Each turn the push reads the owner's words, matches its keys, drops what this reader may not see, puts named items first and the rest above a relevance floor, and places at most five ranked pointers, plus any named items, next to the owner's message. A fuller search runs beside the turn, the agent can still search for itself, and every entry shown goes to the use log. The strip at the foot follows the first request of §1: on its topic key the browser helper scores 0.04, under the floor, and on its use key 0.35, over it.

the skill, as in "Use when working with PDF files" (Anthropic, 2026c): a use key its author wrote. Claude Code's path-scoped rules load when the agent reads or edits a file matching their paths (Anthropic, 2026a), a hand-written situation that triggers a push. For tools, Tool2Vec (Moon et al., 2024) represents each tool by the requests that used it, not by its description, and gains up to 27 points of Recall@K on ToolBench; Re-Invoke (Chen et al., 2024) writes synthetic queries into each tool's document and extracts the intent of a request before matching. T-Mem (Guo et al., 2026) writes *triggers*, anticipated contexts of need, when a memory is stored, after the older idea of indexing a document with the queries it answers (Nogueira et al., 2019). Zou et al. (2026) find descriptive similarity a weak predictor of whether evidence fits a decision (Spearman $\rho = 0.103$ on LoCoMo), the clearest outside support for keying by use.

Memory pushed before the model runs. LangMem's hot-path example searches the store with the user's latest message before every call and puts the results at the head of the prompt, ranked by similarity, importance and strength, which is our fading term (LangChain, 2026a, 2026b; §5.5). MIRIX (Wang and Chen, 2025) keeps procedures and a vault of exact details such as credentials, infers a topic from each input and puts what it finds in all its stores into the system prompt before the model answers: the nearest system that pushes from owned procedures on every turn, keyed by topic and placed in the system prompt, the two choices Find changes. ChatGPT draws on past chats in a new conversation without being asked (OpenAI, 2026); Claude saves its memory as topics the user can read and edit, and searches past chats, as visible tool calls, when a request asks about them (Anthropic, 2026f). OpenClaw, an open-source agent framework, runs a recall sub-agent when the message asks about the past and a quick lookup found nothing strong (OpenClaw, 2026). So the third request of Table 2 is the case today's frameworks already target.

Retrieval over tools and graphs, and feedback. RAG-MCP (Gan and Sun, 2025) retrieves tool descriptions before the model is called: 43.1% tool-selection accuracy against 13.6% for listing every tool. The same retrieval over asset descriptions is our strongest baseline (§9.1). We keep Zep's hybrid of embeddings, BM25 and a graph walk (Rasmussen et al., 2025). HippoRAG 2 (Gutiérrez et al., 2025) links passages through shared entities, which our misses lack; we adopt its rule that an index whose manifest no longer matches the configuration is rejected (HippoRAG, 2026; §8.3). Mem0 takes feedback on each memory (Chhikara et al., 2025; Mem0, 2026); our use log keeps explicit outcomes apart from silence (§6.8).

Each ingredient has a precedent (Table 3). Use keys drawn from requests exist for tools, hand-written ones in skill descriptions and path-scoped rules, and pushes from owned procedures, keyed by topic and placed in the system prompt. In the primary sources we read we did not find them combined: an index of all the owner's assets keyed by use, with keys cut from his corrections, a named slot that yields only to eligibility, pushed where a cache cost model puts it and judged on his replayed corrections. Our search was not exhaustive, so that novelty is unverified.

Table 3 also sets each approach against the morning of §1. A "maybe" means only if stored words happen to match, an item survived a summary, the agent thinks to look, or it is found in a pile of thousands. The lab camera, "last time" and the repeat miss are within reach of products that ship today. The case only Find catches by design is the download, through its use key, and that is where the first deciding test sits (§9.2).

3. Where finding fails

A needed item travels like a parcel. It can be missing from the warehouse (not indexed), carry the wrong label (the wrong key), never be ordered (nobody asked), be held at customs (filtered out), arrive under twenty catalogues (drowned) or sit unopened on the mat (ignored). Customs is meant to stop some parcels, as access control is. Each of these six gates has its own fix and test, so a miss is diagnosed by the first gate it fails. Figure 2 follows the browser helper of §1 through all six.

Gate 1, not indexed. The item lives in a store the index does not cover, such as a catalogue of skills or a note about a machine. No ranking finds what was never indexed. *Fix:* the asset index (§4). *Test:* for each replayed miss, is the item in any index at all?

Gate 2, the wrong key. The item is indexed under what it is about, and the request asks for what it does. A request with no noun ("the one we used last time") fails here too. *Fix:* use keys, co-access edges and a task-reference state (§5, §6.1). *Test:* the item's rank under topic keys alone against its rank with the other keys.

Gate 3, nobody asked. The index could have found the item, but the agent did not search and no push ran, or it ran on the wrong text. The need often appears mid-task, when a tool call fails or the agent is about to say it cannot. *Fix:* a push before the model runs, built from the owner's words, and again on a failed call or a reply that says it cannot (§6.1). *Test:* the share of replayed cases where the item is in front of the model before its first call.

Gate 4, filtered out. The item was found and then removed by the filters that run before ranking (§8.1). Access, quarantine and privacy are meant to remove things; validity is a default, and a wrong mark behind it makes the item vanish from every search with no error. *Fix:* hide on validity only what the writing stage's judge closed (J14 §7.5), and count what each filter removed. *Test:* the hidden share per query, read from the receipts, and an audit of a sample of hidden rows.

Gate 5, drowned. The item was pushed as the fourteenth of twenty old documents, or in a list that is always full, which proves nothing about relevance. Models use information in the middle of a long context less well than at its start or end (Liu et al., 2024). *Fix:* atomic entries,

Table 3. Ten ways to bring memory into a turn, against the morning of Table 2; next week is the same miss after the owner's correction. Cells are reasoning from each design, not measurements. For Find, "last time" holds when the task reference resolves right (§6.1), and next week when the key cut from the correction clears the floor (§5.5), with a pull as the fallback. Costs in base input tokens with the values of Table 5 (§7).

Approach (example)	What reaches the model	Download	Lab camera	"Last time"	Next week	Extra cost	Credit where due
Nearest chunks pushed (plain RAG; LangMem)	The five chunks nearest the request	No	Maybe	No	Maybe	10,600 in the tail; 115,000 a turn at the head	Nothing to curate
Running summary (compaction)	A few hundred words on what happened	Maybe	Maybe	In session	Maybe	Small, cached	Keeps the thread of the task
Extracted facts (Mem0)	A few facts nearest the request	No	Yes	No	Yes	Small	Compact; feedback on each memory
Saved lists (ChatGPT, Claude, Claude Code, skill descriptions)	The same list every session	Maybe	Maybe	No	Yes	The list, cached	The owner edits it; "use when" is a use key
Recall from past chats (ChatGPT, Claude)	Details from past chats	Maybe	Maybe	Yes	Maybe	Not published	Covers every past chat
Path-scoped rules (Claude Code)	A rule, when a matching file opens	No	No	No	Maybe	Only when it fires	A hand-written situation key
Tool search (Anthropic)	Tool definitions the model searched for	Maybe	No	No	No	One search, about 11,000	49% to 74% on MCP evaluations
A bigger window	Everything	Maybe	Maybe	Maybe	Maybe	30,000 a call for 300,000 tokens of memory	Nothing unindexed; simplest
The agent searches (Letta, Claude Code, Managed Agents)	What the agent looks up	No	Maybe	Yes	No	About 11,000 a search	No tokens on guesses
Find (this design)	0 to 5 pointers plus named items, after the cached prefix	Yes	Yes	Yes	Yes	850 for five pointers; 200 in a turn-scoped slot	Costs an asset index, keys, a ranking loop, §9's tests

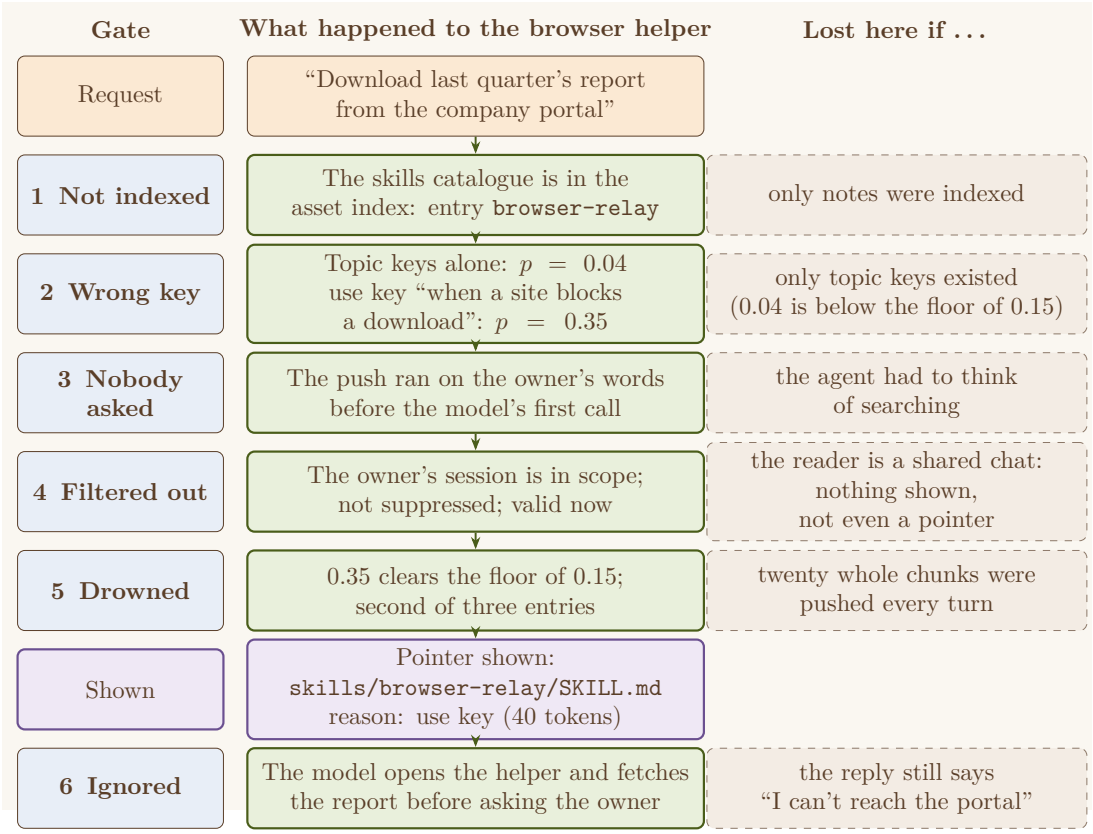


Figure 2. One item through the six gates, for the request "download last quarter's report from the company portal". The item is the browser helper, whose description never says "download". Each row names the gate by how an item is lost there, what happened to this item, and what would have lost it.

a relevance floor, an empty list when nothing clears it, and a slot for named items (§6.3, §6.4). *Test:* the outcome labels of shown entries (§6.8).

Gate 6, ignored. The pointer was in front of the model and the model did not act on it: the reply still says it cannot reach the portal. *Fix:* each entry says why it was pushed (§6.3), and an entry shown and not used counts against its rank (§8.5). *Test:* run the model on each replayed turn and score whether it followed the expected pointer before asking the owner (§9.1).

4. Two indexes

Find keeps two indexes: one over text, which finds what a request is about, and one over what the owner owns, which can find what an item is for.

4.1 The text index

The text index covers the session record kept by the keeping stage and the facts written by the writing stage. It is hybrid: BM25 over words, embeddings over meaning, exact match over names. Each unit is a file or a section, whose shape the writing stage decides, and each row carries the ID and revision of the fact it shows. Find adds an address map to every unit (headings, line ranges, and symbols for code), so a second visit goes straight to the region it needs: *retrieve coarse, address fine*, with units large enough that related text arrives in one read (§7.1). A session the keeping stage flags as temporary leaves no lasting row here or in the use log (J1 §4.7).

4.2 The asset index

The asset index holds what the owner owns, one entry per thing:

- skills, procedures, tools and repositories the agent can run or work in;
- machines and services, with where each is and where its credentials are kept, never the credentials;
- decisions, with their date and reason;
- past approaches: how a kind of problem was solved last time, and where that work lives;
- the current version of an artefact: which model, document or build is the latest;
- rules the learning stage has moved out of always-loaded files (J5 §7.2), found like any other asset and not only when named;
- past sessions, one card each, written by the learning stage's night pass (§6.9).

An entry is short; Figure 3 shows an example entry for the browser helper of §1. Pushed as a pointer it costs about 40 tokens. Entries are generated from the catalogues themselves, a skill's own description or the header of a note about a machine. The file is the ground truth: the index is a cache over the files, rebuildable from them at any time.

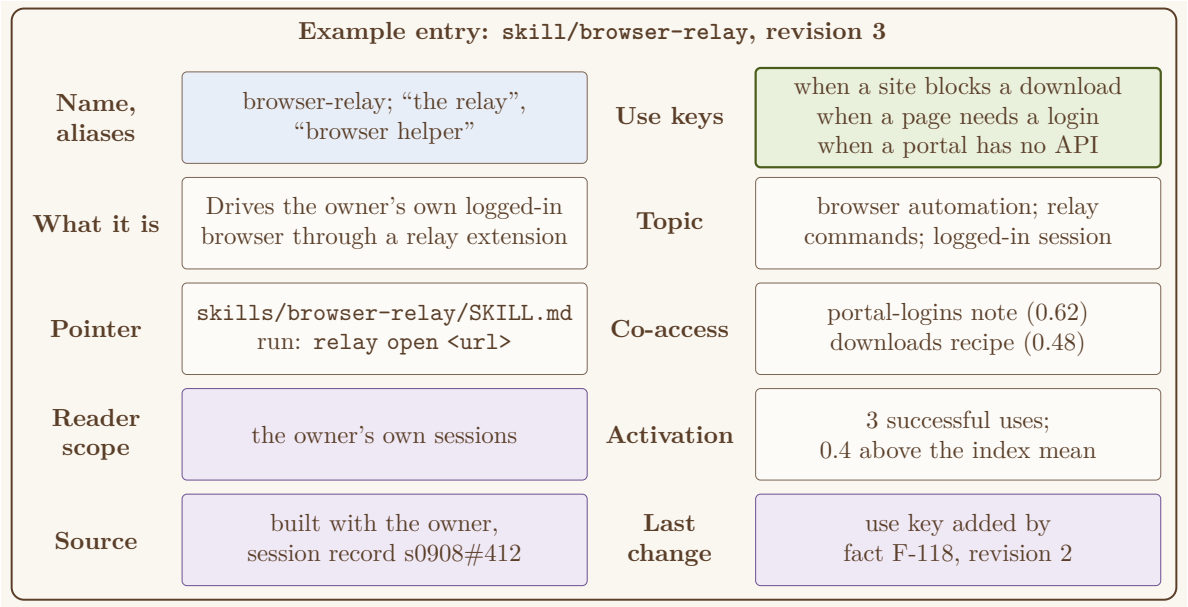


Figure 3. An example asset entry for the browser helper of §1 (illustrative values). The header gives its entry ID and revision; the pointer gives the file. Name and aliases feed the named slot; the pointer says where it lives and how to run it; the use keys say when it helps, in the words of the session that built it and of one later correction; co-access neighbours were fetched with it in the same tasks, with their normalised PMI between 0 and 1 (§5.5); the reader scope is set at indexing (§8.1).

Assets get their own index because the unit is a pointer, the numbers are small (a few thousand entries, in memory) and only an asset can take the named slot (§6.4). A credential's pointer carries the credential's own reader scope (§8.1).

4.3 The index file

The entries needed most often belong in the index file, which is always loaded: one pointer per line, with a short hook saying when to read it. The writing stage owns the shape of a line and checks every write to it (J14 §9.3). Find chooses which lines go in and in what order, the entries most often used without an explicit failure first, up to the budget the learning stage sets (J5 §7.1). One exception: a rule the learning stage moved out of the always-loaded files keeps its

pointer line while its area is active, meaning that a request in the last 30 days matched one of its use keys (J5 §7.2). The per-turn push covers the long tail.

A line the owner wrote by hand stays where he put it (J14 §5.2). A rebuilt file is published only if every pointer resolves, no generated line carries rule text and the file fits its budget; otherwise yesterday's file stays and the build receipt names the failed check. The diff goes to the owner in the learning stage's morning report (J5 §7.5), and a line that carries a rule needs his approval (J5 §7.4). The file sits inside the cached prefix, so a running session keeps the file it started with and takes a new one only at its next start or at a compaction, when the prefix is rebuilt anyway: the night's publish never rewrites a live session's cache.

5. Four keys

A library files a book under its subject; a good workshop labels each drawer by the job it is for. The browser helper's note files it under "browser automation; relay commands", its subject, which is what a topic key matches. The owner's request names a job, "download the report". A use key is the drawer label: *when a site blocks a download*. A name key is asking for the tool by name, which always works if you are allowed to have it. A co-access edge links two tools that come out of the drawer together, here the portal-logins note and the downloads recipe of Figure 3. A key is a way to reach an entry, and each kind catches misses the others cannot.

5.1 Name keys

Name keys are exact names and aliases, a skill, a machine, a project, a nickname the owner uses; they feed the named slot (§6.4), and a name the owner corrects the agent on is added.

5.2 Topic keys

Topic keys say what the entry is about: BM25 and embeddings over its description and body, standard retrieval for requests that do name the topic.

5.3 Use keys

Use keys say when the entry helps, as short phrases: *when a site blocks a download*, *when you need to reach the lab machines*, *when a training run has to be repeated*. They follow encoding specificity (§1): store, with the memory, the cue that will be present when it is needed. They come from three sources.

1. *At write time.* The writer adds up to five use keys to a fact as it stores it (J14 §9.4); the build that generates an asset entry writes them from the catalogue and from the session that produced the asset.
2. *From the owner's corrections.* When the owner says "you have a skill for this", the entry he meant is the one the agent uses next, in that turn or the one after. The key is the situation cut from the request he made just before (*when a site blocks a download*, from "Download last quarter's report from the company portal"). The request itself is kept as the key's source and indexed as a second key text, so a repeat of it matches in full, the way Tool2Vec keys a tool by the requests that used it (§2). These keys are cut from real misses.
3. *Offline.* At night, sessions where an entry's use was a verified success propose keys from the request that led to it.

Generated keys drift toward the generic (*when working on a project*). So each key is tested against past requests: one that would have matched more than one past request in fifty is

dropped, whichever entry it sits on, which also stops a planted key that fires on everything. Each entry keeps at most five. Every key records the addresses it was cut from, its dependency links, which set its readers (§8.1), and it goes when a source is erased or quarantined.

5.4 Co-access edges

Co-access edges link entries fetched together for one request more often than chance predicts, measured by pointwise mutual information (Church and Hanks, 1990). A prompt file, the payload that uses it and the report it writes may share almost no vocabulary and still be fetched together almost every time; no similarity measure proposes that link. Edges are computed per lane, a stream of sessions of one kind (the owner's chats, scheduled jobs, sub-agents), so automated jobs do not swamp interactive sessions, and drive a one-hop expansion from the entries the other keys found. Co-access sees what was fetched, not why, so merge and split proposals go to the writing stage, which owns the shape of files (J14 §9.2).

5.5 Combining keys: one score

Each candidate e gets one number, p_e: the probability that, if shown, e is used in this turn and helps. The event is a helpful use, as the use log labels it (§6.8); the horizon is this turn. The score is logistic in the key matches and the entry's activation:

$$z_e = \beta_0 + \beta_{\text{use}}\, m_{\text{use}}(e) + \beta_{\text{top}}\, m_{\text{top}}(e) + \beta_{\text{co}}\, m_{\text{co}}(e) + \lambda\,(A_e - \bar{A}), \qquad p_e = \frac{1}{1 + e^{-z_e}}.$$

Scorers differ in scale, so for use and topic keys the best score s between the query and the entry's keys of that type becomes a strength m from 0 to 1. The anchors are s_0, beaten by only 5% of random query–entry pairs, and s_1, a clear match. Below s_0 it is 0, above s_1 it is 1, and linear in between:

$$m = \min\Big(1, \max\big(0, \tfrac{s - s_0}{s_1 - s_0}\big)\Big).$$

For cosine similarity and BM25 alike, s_1 is the median score of true matches in the development replay (§8.5), or in the synthetic suite's development half until the replay has enough (§9.4); a cosine of 1 would need identical text. The anchors of each key type and scorer live in the index manifest (§8.3); a build with $s_1 \le s_0$ drops that scorer and says so there. Co-access scores how much more often two entries are fetched together than chance predicts, from 0 (chance) to 1 (always together): for the strongest edge from an entry a the other keys found, $m_{\text{co}} = \max\left(0, \ln \frac{p(a,b)}{p(a)\,p(b)} / -\ln p(a,b)\right)$, with p the share of requests that fetched a, b, or both. Names, and entries resolved from the task-reference state (§6.1), skip the score and go to the named slot.

Starting values. Before any data, $\beta_0 = \text{logit}(0.02) \approx -3.9$: about one retrieved candidate in fifty is used. $\beta_{\text{use}} = 3.3$, $\beta_{\text{top}} = 2.5$ and $\beta_{\text{co}} = 1.5$, so that a full use-key match alone gives $p \approx 0.35$, a full topic match 0.20 and a co-access neighbour 0.08; $\lambda = 1$; and a relevance floor $\tau = 0.15$. The floor is set by gate 5, not by cost (one pointer pays if it saves a search in one turn in seventy, §7.2): a list that is mostly noise teaches the model to skip it. These values are assumptions, re-estimated by the ranking's own loop (§8.5).

The requests of Table 2 come out as follows. The browser helper has $m_{\text{use}} = 0.8$, $m_{\text{top}} = 0.1$ and $A_e - \bar{A} = 0.4$, so $z_e = -0.61$ and $p_e = 0.35$; on topic keys alone, $p_e = 0.04$, below the floor. With anchors of, say, $s_0 = 0.3$ and $s_1 = 0.6$, that use-key match is a cosine of 0.54; the encoder the manifest names measures the real ones. The lab machine is named, and "the setup we used last time" resolves through the task-reference state, so both go to the named slot without a score. A repeat of a corrected request matches the request kept with its key in full, so at the index's mean activation the entry scores 0.35 and is pushed.

Fading. The odds that an item is needed again fall as a power of the time since each past use, summed over those uses, and human memory follows the same curves (Anderson and Schooler, 1991). So each entry carries an activation from its own use log:

$$A_e = \ln \sum_j \max(t_j, t_{\min})^{-d},$$

In plain terms, with $d = 0.5$, each past use adds one over the square root of the hours since it: a use an hour ago adds 1, a day ago 0.2, a year ago 0.01, and the activation is the log of the sum. Here t_j is the time in hours since the j-th use of e not labelled an explicit failure, the floor $t_{\min} = 1$ hour keeps a use a moment ago from giving an infinite activation, and d starts at 0.5, the default of the ACT-R theory built on that finding (Anderson et al., 2004). An entry never used counts its write as one use; $\bar{A}$ is the mean over the index at the last build. With $\lambda = 1$, fading can outweigh a key. In an index whose mean is two uses 1,000 hours ago, an entry used once, a year ago, sits 1.8 below the mean, which takes a full use-key match from 0.35 to about 0.09, under the floor. The writing stage marks importance when it writes a fact (J14 §8.5); this stage decides how it acts at read time.

Fading lowers a rank and never removes an entry from the index: the named slot ignores it and the pull still finds it. For entries marked important, corrections and decisions, the term is floored at zero, $\lambda \max(A_e - \bar{A}, 0)$, because a correction is needed when the old mistake comes back, perhaps months later; such an entry still has to match a key to enter the list. Fading is about need, not truth: a fact that stopped being true is closed by the writing stage's judge.

6. The turn: push and pull

Picture three assistants. One waits until you ask whether you already have something for this: the pull. One piles every file you own on your desk each morning: a bigger window. One reads your request over your shoulder and puts a sticky note on top of the pile, "the browser helper, for blocked downloads": the push. Find is the third, and its note always goes on top of the pile, never into the middle, which is what keeps the cache (§6.5).

The push is the only part of the design that works when the agent does not suspect it has something (gate 3). Appendix B gives it as pseudocode.

6.1 The query, "last time" and a failed call

The query is the owner's words, read from the `owner_words` field the keeping stage fills when a turn is appended (J1 §4.1), never the wrapper the harness adds around them (timestamps, channel headers, injected context); *test:* wrapped prompts must produce the same list as unwrapped ones. For full-text search it is the tail of those words, at most 2,000 characters, since the request usually sits at the end of a long paste. A session with no history of its own, a new sub-agent or a fresh chat, adds to its first push's candidates the latest entries of the owner's previous session that it may read.

"Last time." The owner's words cannot resolve "train it again with the setup we used last time" (Table 2): the retriever does not see the window the model sees. So each session keeps a small **task-reference state**: the title of the task in progress (from its first request until the night pass writes the session card), the IDs, never text, of up to twenty entries resolved or used in its recent turns, and any reference still open, updated from the use log after each turn. A mention with no noun is spotted from a short list of phrases per language ("last time", "that setup", "as before"). It resolves first to the most recent entry of the kind it names (a setup, a file, a machine) in this session, then to the most recent session card whose task title matches the request's task words. When two fit about equally, both go in one line of names and the

agent asks; a guess never takes the slot. A single resolved entry takes the named slot (§6.4), after passing eligibility like any other (§8.1). Its pointer says how it was read (" 'last time': the session of 3 October"), and the agent repeats that in its reply, so a wrong reading shows before the retrain starts. *Test:* wrong readings are counted (§9.2).

When a call fails, or the agent is about to say it cannot. The need often appears mid-task: the portal refuses, and the agent is about to write "I can't reach the portal". So the push runs again on a failed tool call: an error the tool declares, a status outside the success range, or a login or captcha page recognised by its type, since many portals answer a blocked download with a login page and a success code. The query is the tool's name, its arguments and the kind of failure, never the free text of the result, so a fetched page cannot choose what gets pushed, and the list goes after the tool result. The push also runs once on the agent's draft reply when the draft declares it cannot do something ("I can't", "I have no credentials") or asks the owner to choose ("which setup?"), before the reply is sent. Its query is the kind of claim plus the owner's words, never the draft's own text. All three misses of §1 show in such a reply. It costs one more model step on the few turns where it fires, and gives the owner a rule he can see: the agent never says "I can't" without checking what it has.

6.2 What may run on the path

The push has a budget: 50 ms at the 95th percentile from the owner's words to the packed list, split into targets to be measured (Appendix B, §9.2). The query is always new, so the encoder gets 15 ms of it: a small local sentence encoder of the Sentence-BERT family (Reimers and Gurevych, 2019), the same model that built the index, named in the index manifest (§8.3). If it misses its budget, is absent or does not match the manifest, the push falls back to names and BM25 over use and topic keys, and its receipt, the record every step leaves (§8.4), says "lexical only". A full search of the text index, over the bodies of notes and session records, does not fit; it runs beside the turn, and its hits join the next turn's candidates, scored like any other and marked as found for the previous message.

An owner who writes in more than one language needs a multilingual encoder: a key cut from a correction in one language has to match a request in another, which BM25 cannot do. The lexical fallback then finds only keys in the request's language, and its receipt says so.

6.3 The list

The list holds up to five entries with $p_e \geq \tau$, plus any named items. Each entry carries its ID, revision and the key that matched, which helps the model judge it and lets the replay attribute a hit to a key. An entry goes in whole or as a pointer by the line §7.3 derives, and the whole list is capped at 1,000 tokens a turn: an item that would go in whole past the cap goes as a pointer. When nothing clears the floor the list is empty, and an empty list is a correct answer.

Table 4 sets the list against what reaches the model today for the first request of §1. The contents are described, not printed.

6.4 A guaranteed slot for a named item

"*Check the camera on the lab machine*" names the machine. When the owner's words name something the name table knows, a skill, a machine or a project, its entry goes in first, whatever its score. The named case is where a miss annoys most ("I told you its name") and the cheapest to get right, and a line in an always-loaded file telling the agent to check its catalogue does not bind reliably, so the slot is a check in code.

The slot yields to one thing: eligibility. Names are matched against the reader's authorised view of the name table, and a named entry then passes the same eligibility steps as every other candidate before any slot is reserved (§8.1). A named item that fails any step takes no slot and

Table 4. The first request of §1, before and after: what a pointer is, why the list is short and where the saving comes from.

	What reaches the model after the cached prefix	Cost (Table 6)	What the agent does
Nothing pushed (today)	The owner's message only	Nothing	Tries the portal, meets a login page, answers "I can't reach the portal"
Plain RAG	Five chunks nearest the request, about 2,500 tokens: a session excerpt about an expired vendor link, the note on where downloads are saved, a report template, two meeting notes that mention the portal. The browser helper is not among them	10,600 base tokens over their life in the tail; 115,000 a turn at the head	Likely the same answer
Find	One line of about 40 tokens: the browser helper's entry and revision, "drives the owner's logged-in browser", the key that matched (*when a site blocks a download*) and the file to read. Nothing else clears the floor	170 base tokens over its life, 0.07 cents	Opens the helper and fetches the report

leaves no hint that it exists. Up to three eligible named entries take slots, in the order named; further ones share one line listing their names. Named entries never take one of the five ranked places. A name that is also an ordinary word (a skill called "summarize", a machine called "lab") takes the slot only when the owner's words mark it as a name, by quotes, capitals or code formatting, or when a use or topic key of the same entry also matches, as the lab machine's note does for a request about its camera. *Test:* every eligible named case gets its entry, every time; a named item outside the reader's scope, quarantined, suppressed or erased exposes nothing; and the negative set of §9.1 reports the named entries pushed when nothing was needed.

6.5 Where the list goes

The prompt cache works like a long document the provider keeps typeset: change a page in the middle and every page after it is typeset again; add a page at the end and only that page is. So the list goes after the cached prefix, next to the owner's message, never into the system prompt or an always-loaded file. Models also use information at the end of a long prompt better than in its middle (Liu et al., 2024). §7.4 prices the wrong place.

Where the provider offers a turn-scoped slot, the list goes there. Anthropic's API takes a system message marked to clear at the next user message: shown for this turn, it then stays in the conversation empty, costs no input tokens and keeps the cache matching (Anthropic, 2026d). Where there is no such slot (other providers, and command-line tools that own the window) the list goes into the persistent tail next to the owner's message. The tail also carries the learning stage's provisional rule lines and correction sentences still waiting to be processed (J5 §6.1), priced the same way.

6.6 The pull

A search tool over both indexes is part of the agent's base tool set. It answers in a second or two, uses the same keys and filters, returns pointers first, and has a range mode that returns the session-record entries a compaction marker names (J1 §8.1). At most N pointers come back (10 to start), each clipped to 300 tokens to start, so one large hit cannot hide the rest, and the answer says how many further matches exist, so a clipped answer never reads as complete. A

pull registered but not offered to the model does not exist for the agent, so §9.5 checks it is offered under every window controller, whoever decides what stays in the model's context: the runtime, the provider's API or the provider's command-line tool (J1 §6.1).

6.7 What the context already holds

An index answers *where is it?* A read ledger answers *do I already have it?* Before a read, it checks whether that file region is already in the window and unchanged. A matching hash proves the file did not change, not that the text is still in the window, so the ledger reads the eviction events the keeping stage publishes after each compaction or rebuilt window (J1 §6.6) and drops every entry whose range left. When the provider's API or command-line tool compacts the window itself, nobody can tell what is still in it, so the ledger drops every entry older than the event. A read of a path that does not exist is logged as a lookup miss for the writing stage (J14 §9.2).

6.8 The use log

For each entry shown, by push or by pull, the log keeps one row. *Shown* records that it was sent to the model, and how sure the window controller is that the model received it: known input, or sent but unconfirmed, when a provider's API or command-line tool manages the window and may compact or drop the item before the model reads it (J1 §6.6). *Used* means the reply referred to it or a later tool call followed its pointer; *body read* means a pull fetched a body not already in the window (§7.1). A used entry then gets one of four outcome labels. **Verified success**: something observed confirms it helped, such as the pointed-to tool call succeeding at the step it was fetched for, the owner's explicit approval, or a sampled audit. **Explicit failure**: the owner says it was wrong, outdated or not the one he meant, or its pointer was broken. **Unchallenged use**: used, and nothing confirmed or contradicted it. **Unknown**: the session ended before an outcome could be seen. Silence is never success. An entry fetched with the pull that the push had not shown is labelled *missed by push*, a fact about the push and not a success. The ranking learns from these labels with their uncertainty (§8.5).

The log answers which memories were supplied or cited for an answer, by ID and revision: attribution, not causation, least of all for input sent but unconfirmed. An item the owner asked not to mention joins a suppression list read at the privacy step (§8.1): it stays stored and visible in his own view and is no longer pushed or returned.

The owner's inspector, one owner-only view over these records, shows each item's sources and state and keeps three actions apart: correct, don't mention and delete (Appendix D).

Tests, health probes and other synthetic calls run under their own identity and never write to this log.

6.9 Ask the session that did the work

People in a team do not remember everything; they remember who knows. Wegner (1987) called this transactive memory, and it gives Find a fourth route: the session that did a piece of work, which read the files, weighed the options and knows why it chose, more than any note written afterwards keeps.

The learning stage's night pass writes one card per finished session (J5 §6.2): what it decided or built, the files it touched, its dates and its reader scope. Cards are indexed like any other asset, with use keys. When a card matches the turn, the push carries a pointer, "session s0412 worked on this, 3 October", and a pull can ask it. Asking means searching that session's record first; if the record answers thinly, the session itself is resumed, not copied, with one question and a note saying why it was woken (J1 §8.3). Its answer is testimony from its own time, labelled

Table 5. Notation and prices for the series, in multiples of the base input price except π_s and π_ℓ (dollars). The rows below the heading serve the companion papers.

Symbol	Meaning	Value in the examples
r	price of reading one cached token	0.1 (0.05 or 0.025 on some models)
w	price of writing one token to the cache	1.25 (five-minute cache); 2 (one-hour cache)
g	price of one output token	5
o	output tokens of one tool call	about 100
L	tokens already cached in the context window	100,000
P	position of the first token that changed	20,000
x	tokens returned by one read	500
B	tokens of an item's body	varies
B_p	tokens of a pointer to it	40
M	later calls in the session that re-read the context	30
q	probability of a body read: a pull fetches the item's body, not already in the window, at least once in the rest of the session (horizon: the M later calls)	0.1 to 0.5
p_e	probability that entry e, if shown, is used and helps (horizon: this turn; §5.5)	from the score
L_s, L_a	cached context of a past session and of the session asking it (§7.5)	per session
	Used by the companion papers	
q_{turn}	probability that a later turn pulls from an evicted span (horizon: one turn; J1 §7.2, the price of moving the prefix)	not yet measured
H	turns a pulled slice stays in the window (J1 §7.2)	not yet measured
c	input price of the model that writes a marker (J1 §7.2)	not yet measured
K	cost of one marker of m tokens over S tokens, $K = cS + g\,m$ (J1 §7.2)	per span
q_{90}	probability that some session in the next 90 days needs a detail (J14 §4)	estimated per detail
π_s, π_ℓ	base input price of a small and a large model (Claude Haiku 5.5, Claude Opus 5.5), in dollars per token	\$0.10 and \$4 per million

with its dates. Reader scope binds here as everywhere (§8.1): a narrower reader never wakes, or learns of, an owner-private session. §7.5 prices a wake against a search.

7. What finding costs

7.1 A read costs a turn

This section holds the notation and prices for the whole series (Table 5): symbols another paper reuses are registered here; symbols local to one section are declared there. Prices are Anthropic's published ones (Anthropic, 2026b), and each probability names its event and its horizon.

Every trip to the archive pays the same toll at the bridge, a re-read of the whole context, whether it brings back one page or ten; so the design saves trips, not pages. A pull is one more model call. It re-reads the whole cached context, writes the x tokens it returns into the cache,

Table 6. What each choice costs, as arithmetic on Table 5 at $4 per million base input tokens. A cheaper cache read cuts the search, tail and large-window rows by a third to a half; the rewrite rows barely move.

Choice	Base tokens (read 0.1)	Cents (read 0.1)	Cents (read 0.05)
One search (a pull), its result re-read over 30 calls	12,600	5.0	2.8
One pointer in the tail, over its life	170	0.07	0.04
Five pointers in the tail, over their life	850	0.34	0.22
Five pointers in a turn-scoped slot, one call	200	0.08	0.08
Five 500-token chunks in the tail (plain RAG)	10,600	4.3	2.8
The same chunks at the head of the prompt, every turn	115,000	46	48
A 1,000-token block a fifth of the way in, every turn	92,000	37	38
300,000 tokens of memory in a large window, every call	30,000	12	6
One ownership miss: the owner's "you have this" turn and the search it starts	about 23,000	9	5

and pays for its own output:

$$C_{\text{pull}} = rL + wx + g\,o.$$

With the values in Table 5 this is about 11,000 base tokens, of which the size of what was read, wx, is only 625. A read costs a turn, not bytes. So count reads: a unit that brings related text in one read saves turns (§4.1), and a pointer pushed in advance removes the search that would come before the read.

A helpful use does not imply a body read: a pointer may be all the turn needs, or the body may already be in the window (§6.7). So the log records body reads apart from uses (§6.8), and q is estimated per kind of entry as the share of shown pointers followed by a body read; until it holds those counts, the push uses p_e as a stand-in, an assumption the estimate replaces.

7.2 What a push costs

The list can sit in one of two places (§6.5). In the persistent tail, an item of B tokens is written once at w, then stays in the session and is re-read at r by each of the M later calls:

$$C_{\text{push}} = B\,(w + rM).$$

With $M = 30$ the factor is $1.25 + 3 = 4.25$, so a 40-token pointer, such as the browser helper's, costs about 170 base tokens over its life and a list of five about 850. A pull costs about 11,000 plus another 1,500 for its result re-read over the same M calls. In the tail, the list pays for itself if it saves a search in more than about one turn in fifteen.

In a turn-scoped slot the item cannot be cached, so it bills at the base price, B for each call that sees it, and nothing after its turn. Five pointers then cost 200 for a turn of one call, and the list pays for itself if it saves a search in about one turn in sixty. The slot also removes a cost the tail formula leaves out: thirty turns of a persistent tail leave thirty old lists in the context, and the model can still follow a pointer from turn 3. Its limit is the tool loop. A tool result counts as a new user message and clears the slot, so a list that must stay in view through k calls is sent again after each result, at kB.

Table 6 puts these prices in cents at a large model's base input price of $4 per million tokens (Table 5): once at the read price of 0.1 used in the examples, once at the 0.05 of Claude Opus 5.5, the model that price belongs to.

The list is paid on every turn and a miss happens on a few turns a week, so the numbers to carry away are rates. In tokens alone, five pointers in the tail pay for themselves if they prevent a miss about once in 27 turns, or save a search once in 15; in a turn-scoped slot, once in 115 and once in 63. At Opus 5.5's read price the tail list pays at one prevented miss in about 23 turns. The tokens leave out two larger terms: a miss costs the owner's attention, a sentence he should not have had to say, and at least one more turn; and a pull costs seconds against the milliseconds of an on-path lookup.

7.3 Whole or pointer

An item whose body has B tokens can go in whole or as a pointer of B_p tokens. Write $F = w + rM$ for the life cost of one pushed token and $C = rL + g\,o$ for the turn a pull costs. Whole costs BF. The pointer costs B_pF and, with probability q, a pull that also writes the body and leaves it to be re-read, an expected $q\,(C + BF)$. Both are counted over the same M later calls, as if any pull came in the turn the pointer arrives; a later pull re-reads a longer context but leaves the body in place for fewer calls, two opposite effects we do not model. Going in whole wins when $BF < B_pF + q\,(C + BF)$, that is, when

$$B < B^* = \frac{B_p}{1-q} + \frac{q\,C}{(1-q)\,F}.$$

With the values in Table 5 ($F = 4.25$, $C = 10{,}500$), B^* is about 2,551 tokens for an item with $q = 0.5$ and about 319 tokens for $q = 0.1$; at $q = 0$ it falls to the pointer itself. As q approaches 1, B^* grows without bound: an item certain to be needed goes in whole at any size, and one rarely needed goes in whole only when it is about as short as its pointer. A longer context raises B^*, a cheaper cache read lowers it (Figure 4a). The line holds for the persistent tail. An item pushed whole into a turn-scoped slot is gone next turn, so a later need pays a pull anyway; there an item goes in whole only when this turn is likely to need it, and the 1,000-token cap of §6.3 bounds the list either way.

7.4 Where it goes

Back to the typeset document of §6.5. If a block that changes every turn sits at position P inside the cached prefix, the next call has to write the $L - P$ tokens after it to the cache again, at w instead of reading them at r. That adds

$$\Delta C = (w - r)(L - P)$$

to every turn. A 1,000-token block a fifth of the way into a 100,000-token prompt ($P = 20{,}000$) costs about 92,000 base tokens a turn, the price of eight searches (Figure 4b); at the end it costs its own write, about 1,250. The bar for placement follows: memory in the tail adds the write of its own tokens and no rewrite of the unchanged prefix (§9.2). The same term prices a one-time eviction from the middle of the window (J1 §7.2).

7.5 Waking a session against searching

A woken session that has left the cache rereads its whole context once, written to cache at w; call that context L_s tokens. The asker's search costs a turn per step (§7.1), each re-reading the asker's own context of L_a tokens at r. Ignoring the answer itself, the wake pays in tokens when it saves more than

$$k^* = \frac{w}{r} \cdot \frac{L_s}{L_a}$$

search calls. On Claude Opus 5.5, whose cache reads cost 0.05 of the base input price, that is $25\,L_s/L_a$: a 270K-token session asked from a 500K-token chat pays after about 14 saved calls, from a 70K-token chat after about 96. At the $r = 0.1$ of Table 5 both thresholds halve, to about 7 and 48. A second question inside the cache lifetime costs only $r\,L_s$. In time, a wake is one call where a search is several, but a call over a long context; which is faster depends on the question. So a long chat should ask the session and a short one search first, and §9.2 measures both on the owner's own questions.

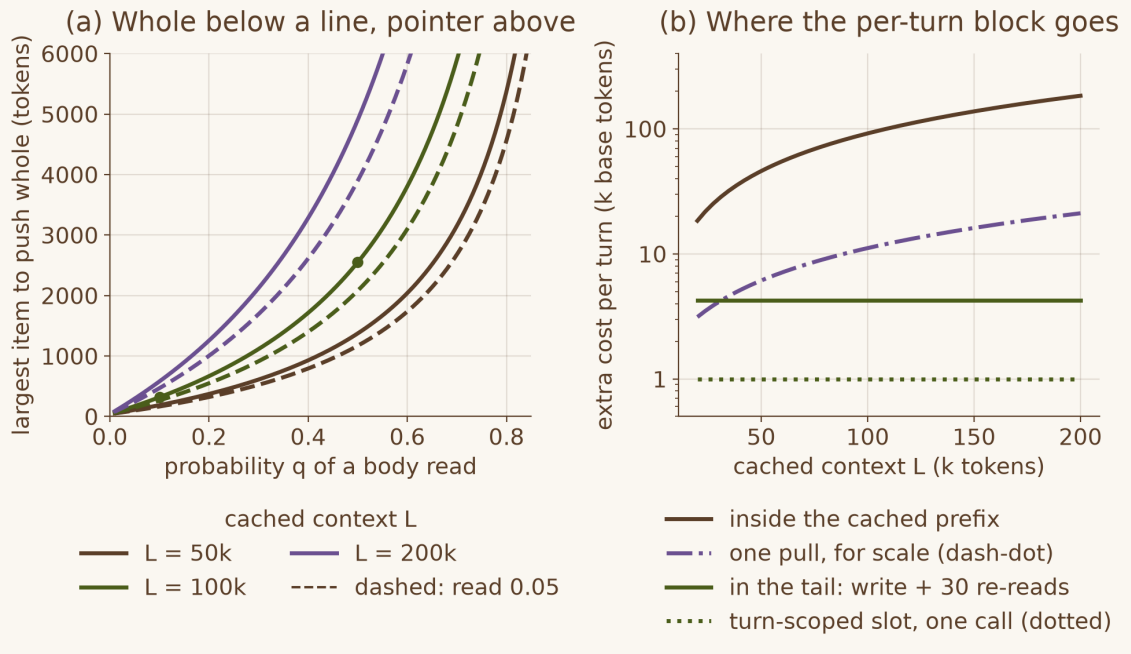


Figure 4. What finding costs, from published cache prices. (a) The largest item worth pushing whole, against the probability q of a body read, for three context lengths; solid lines at a cache-read price of 0.1, dashed at 0.05; a 40-token pointer, $M = 30$. Below a line, push whole; above it, push a pointer. The dots mark 319 tokens at $q = 0.1$ and 2,551 at $q = 0.5$ (§7.3). (b) Extra cost per turn of a 1,000-token block that changes every turn, on a log scale: inside the cached prefix at 20% of the context (the rewrite it forces), in the persistent tail (its write and 30 re-reads), and in a turn-scoped slot (one call, dotted), with one pull for scale (dash-dot).

8. Built ahead, kept current

Most of the work of finding happens before anyone asks. Every accepted write enters the fresh-writes overlay at once (§8.2), and a changed catalogue regenerates its asset entries. Every night a new index is built beside the live one (§8.3), use keys are refined, the index file is published (§4.3) and the ranking is refitted (§8.5); every week, co-access edges are recomputed. The learning stage's night pass works on rules (J5 §6.2) and never touches these.

8.1 Filters, in a fixed order

Ask *"check the camera on the lab machine"* in a chat shared with a colleague, and the machine's entry, which says where its login is kept, must not appear. A private pointer shown in a shared chat or to a sub-agent is the worst daily failure this stage could cause. So every candidate passes one function, `eligible()`, whatever route brought it: ranked, named, fresh-writes overlay or task reference. Ranking and slots come after it, and no later step can undo an earlier one. Figure 5 draws the order.

A fact can have several revisions, each with its own readers, so `eligible()` first picks the revision it would return and checks only that. The revision map (§8.2), holding the latest revision of anything changed since the build, and the date asked about (today, unless another is named) pick it; picking can drop a candidate or swap its revision, never add one. If permissions change mid-query, authorisation runs again at return.

1. **Authorisation.** The reader comes from the authenticated session, never a field of the request: filtering by namespace is not access control (Zep, 2026). Every entry carries a reader scope: the readers allowed to see it. Find sets it at indexing, from where the content came (its source class, J14 §8.1) and where it was written. Anything from the owner's private sessions, and any pointer to a credential, is his alone. Scopes are sets, not levels: each of two scopes can admit a reader the other refuses. So an item built from others (a use key, a co-access edge, a generated entry, a hold, which is a write parked until the owner approves it) is readable only by readers all its sources admit, traced through dependency links (the writing stage owns their schema) and checked at return. That can be nobody. A held rule built from the owner's private correction and a recipe a supplier's sub-agent may read is his alone: the sub-agent gets nothing, not even a count, unless he grants it. The owner may widen or narrow a scope; an agent may only narrow one. Outside its scope an entry does not exist: no list, slot, graph walk or count shows it, and a narrower reader reaches memory only through this filtered service, never the files. *Tests:* its direct file read is refused; a task reference is refused once the reader loses access to its source, even in a query already running.
2. **Quarantine and privacy suppression.** Erased items are gone from the indexes (J14 §7.7), and until the next build the revision map of §8.2 hides any copy left in the base index. Quarantine, which the writing stage decides, marks every row reached through links from the span, and only an audit view shows them. Items the owner asked not to mention (§6.8) stop here too.
3. **Temporal visibility.** *Current-state* resolution, the default, sees what is valid and believed now: the revision map keeps the latest revision, and a fact the writing stage superseded or retracted is hidden (J14 §7.3). *Historical* resolution answers what was true, or believed, at a date (J14 §7.4): it skips the latest-revision rule and picks the revision by valid or recorded time. Steps 1 and 2 bind both, and so do the map's tombstones. *Test:* asked what was believed the day before a correction, the owner gets the old revision; once the correction is erased, no row of it. A fact whose first revision is public and whose second is private returns nothing to a public reader in current-state mode, and only the public revision in historical mode.

A show-hidden mode relaxes step 3 only, showing superseded and retracted rows, marked, and says how many step 3 hid. The counts each step removed go to the receipt, readable only with audit permission, since a count can itself reveal that something exists.

8.2 Becoming retrievable

A correction you make now, such as the "you have a skill for this" of Table 2, must be findable on your next message, although the index is rebuilt only at night. A library has the same problem. It reshelves returned books overnight; until then they wait on a trolley by the desk, and the desk checks the trolley before the shelves, so a book returned at noon can go out again at one. A slip on the old shelf, "withdrawn, newer copy on the trolley", stops the old copy going out, and while a book is being recalled the desk stops lending for a moment and says so. Find has all three: the fresh-writes overlay is the trolley, the revision map the slip, and the epoch fence of §8.3 the pause.

Never "I remembered that": the agent names a correction's status. Each stage owns one: **captured** (recorded, J1), **accepted** (through the writing stage's check, J14; a held write

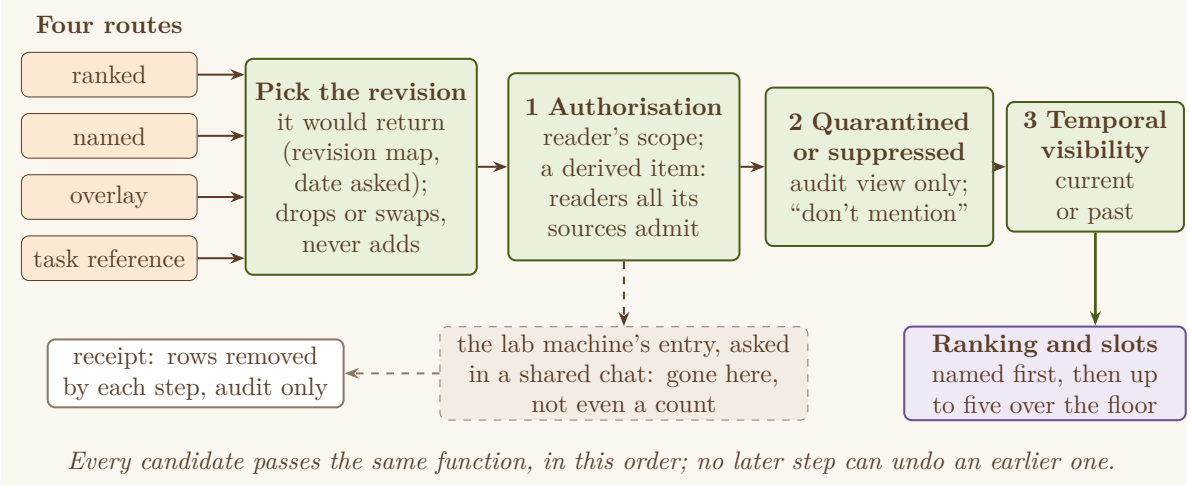


Figure 5. The eligibility check, in the order it runs. Candidates from all four routes meet one function; the revision it would return is picked first, then the three steps remove what this reader may not see, and only what survives is ranked or given a slot. The lab machine's entry, asked about in a chat shared with a colleague, drops out at step 1 and leaves no trace in the list; each step's count goes to the receipt, readable with audit permission.

is acknowledged, not accepted), **retrievable** (here), **active** or **enforced** (J5). They are independent; a correction can be active in the tail before it is retrievable. Retrievable means the next relevant request, even in this session, reaches it through the filters of §8.1: a pull always returns it, and a push carries it when it clears the floor, as a repeat of the corrected request does (§5.5).

The **fresh-writes overlay** is a small table every request checks: the writes since the last build, in memory, with their keys and reader scope, until the next build takes them in. The worst miss of all is contradicting what the agent wrote an hour ago, and the overlay is what prevents it. The writing stage tells Find about each accepted write with a delivery slip and resends it until Find confirms (J14 §7.2). The confirmation makes the item retrievable, and the correction's provisional line leaves the prompt's tail once every part of the correction is retrievable or held (J5 §6.1). Appendix C gives the delivery in detail, including what happens when an erasure crosses it, and Table 12 lists the failure cases with what the agent may say in each.

New text alone cannot stop old text from being served: a query matching only the superseded wording would find the old row in the base index and never meet the new one. So the overlay keeps a **revision map**, an entry for every ID changed since the build with its latest revision or a tombstone (a deletion marker), applied to every base-index candidate before the filters: an older revision or a tombstoned ID is dropped in current-state resolution (§8.1). *Test:* a query matching only the old wording misses the old revision.

A held write is served only as a hold, never as a fact or a rule in force, and only to the readers §8.1 allows. A line in the writing stage's pending area, written while its judge was down, is never merged as a fact: the pull serves it labelled "unchecked" and the push never shows it (J14 §5.3). A row in the learning stage's rule ledger is a proposal, not retrievable as a rule in force (J5 §4.1); a rule line enters the overlay only on the writing stage's receipt.

8.3 Manifests, restarts and rebuilds

The nightly build must never bring back something erased, and it must not lose a correction made while it ran.

Every build writes a manifest when it starts: schema version, encoder and tokenizer versions, the anchors s_0 and s_1 per key type, the snapshot of the store it read, the erasure epoch then, and

its counts (Appendix A). The store raises its epoch, a counter, before any erasure or quarantine (J14 §7.2), so a reader can tell old state from new. A build is swapped in only if its manifest matches the running configuration and its epoch is still current; otherwise it is discarded and rebuilt, never mixed with the live index.

A swap keeps every write newer than the build's snapshot, and no read serves state older than the newest erasure; the push waits at most 10 ms for that, then sends an empty list and leaves the finding to the pull. Figure 6 shows both cases, and Appendix C gives the rules and their tests.

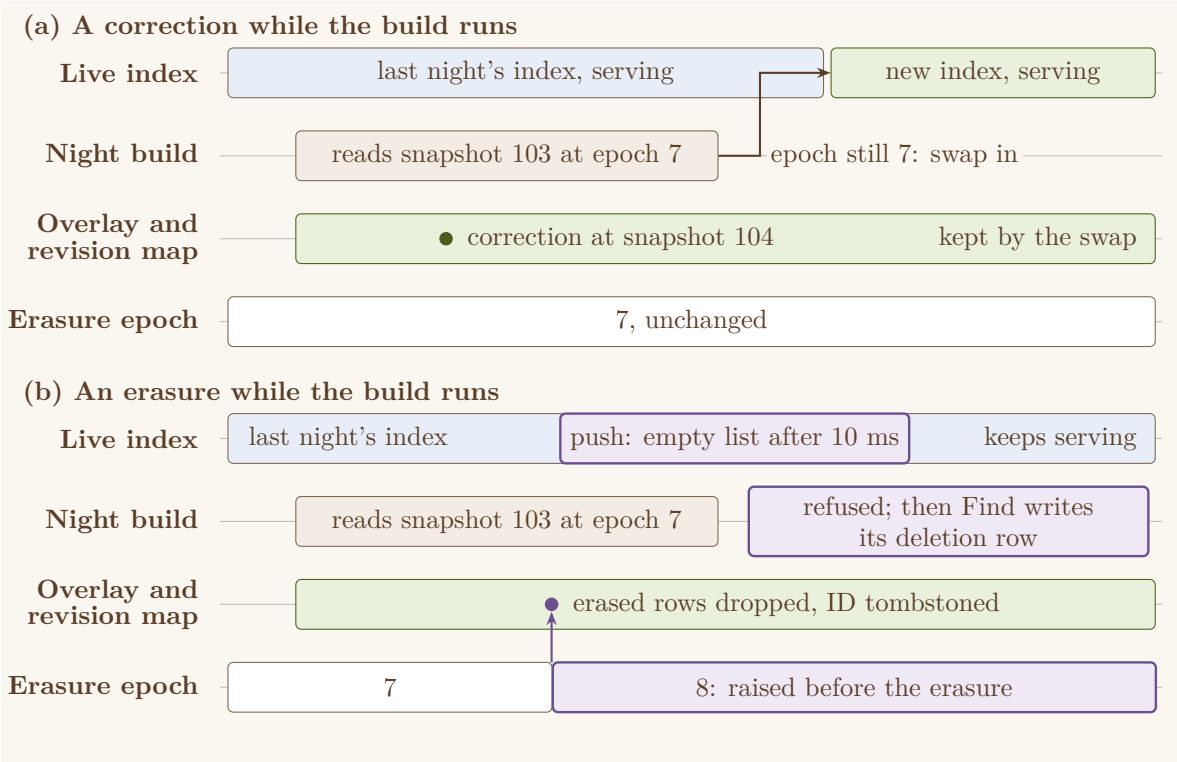


Figure 6. Freshness across a night build, four lanes on one time axis. (a) A correction made while the build runs enters the overlay at once and survives the swap, because the swap keeps every overlay row newer than the snapshot the build read. (b) An erasure during the build raises the epoch: the build is refused, the live index and the overlay keep serving, and Find writes its row of the deletion receipt only after the refusal. A push that arrives while the erasure is being applied waits at most 10 ms, then sends an empty list.

8.4 Receipts, and what a zero means

The series rule that the alarm is a missing receipt or a missed planted opportunity, never a zero count (J5 §9.1), holds here: every turn produces a push receipt, empty or not, and a turn without one is a defect. Appendix C lists what else is watched, and how.

8.5 The ranking's own loop

Every night Find refits the score of §5.5 (the weights β, λ, the fading exponent d and the floor τ) on the entries the use log says were shown. Each shown entry gets a target: 1 for a verified success; 0 for an explicit failure, and for an entry shown but not used; for an unchallenged use, its audit label if the audit sampled it, and otherwise that week's share of audited uses found helpful. Unknown rows are left out. Each week a judge model from another vendor audits 50 unchallenged uses, labelling each helpful or not from the turn, the entry and what followed.

The owner sees ten of those labels a month in the learning stage's morning report (J5 §7.5) and is never asked for more; where he disagrees, his label wins, and the judge's agreement with him is reported with each fit. The fit is pulled toward the starting values, so a key type seen only a handful of times stays near where it began. The log knows only what was shown: a missed-by-push row counts as a positive only when its use was a verified success, and what an entry nobody fetched would have done, only the offline replay can say.

A loop that learns from its own outcomes can overfit them, so three sets are kept apart. The *development replay*, corrections up to a freeze date, is all the nightly fit and any tuning may read. The *regression suite* holds every eligible named case, every fixed failure and the planted opportunities; new weights are adopted only if they lose none of them and do not lower hit at 5 on the development replay. The *evaluation stream*, corrections after the freeze date, is held by a separate evaluator: the loop and whoever tunes it never see its expected items, and the fit leaves out the use-log rows of those cases' turns, from the request through the owner's correction to the turn after. The stream is scored only when a question of §9.2 is decided; its cases then join the development replay and a new freeze date starts (J5 §9.2).

9. How we will know

The main test is built from the owner's own corrections, the misses that matter to him.

9.1 The correction replay

Each case comes from one correction such as "you have X". It holds the owner's words in the request just before it, the expected items, the reader's scope and the date, so the indexes can be rebuilt as they stood then. Cases go into the sets of §8.5. Some come from corrections made mid-task, after a tool call failed, so the push on a failure (§6.1) is tested too. Two raters, working apart, assign every failed case the first gate it failed (§3); a miss that fits no gate is itself a finding.

The owner corrects only what the system running at the time missed, so which arm runs live while cases are collected decides what the cases can show. Replaying an arm on its own misses favours any challenger, and a challenger's losses on requests the live arm got right never become cases. So each stream names its live arm, and every arm is scored on two sets: the live arm's misses, and an equal sample of its verified successes from the use log. The live arm itself is judged only by randomising which arm runs per session and comparing the rate of "you have this" per hundred turns (§9.3). The replay keeps the job it does best: placing each miss at a gate.

The replay holds only misses the owner named. A *negative set* of ordinary turns with no correction after them shows what each arm pushes when nothing was needed, and how often the model *followed wrongly*, acting on a pointer a judge labels unhelpful there. Misses he worked around in silence stay invisible (§10).

Eight arms run on the same cases. Arm A is the ablation base, the shape of the hot-path systems in §2; the catalogue arm is the strongest asset-aware design we know of.

Each arm reports hit at 5 (the expected item is in front of the model before its first call; an item already in the always-loaded index file counts, and is reported apart) and, because delivery is not use, *followed*: the model is run on the replayed turn and scored on whether it followed the expected pointer before asking the owner (gate 6). It also reports followed wrongly on the negative set, pulls, tokens pushed, empty lists, outcome labels, explicit failures, cache writes and push latency.

Table 7. The arms of the correction replay. A is the ablation base; the catalogue arm is the competitive baseline.

Arm	What runs
A (base)	Topic keys over the text index; five whole chunks pushed each turn
B	A, plus the asset index with name and topic keys
C1	B, plus use keys written by authors and by the build: a skill's "use when" text, keys written with each fact
C2	C1, plus use keys cut from the owner's corrections
D	C2, plus co-access edges and the task-reference state
E	D, plus the named slot
Catalogue	Every asset's name and full description, "use when" text included, retrieved by the request with the same hybrid search and filters, the top lines pushed at E's token budget and within its latency budget (§6.2)
Pull only	No push; the agent has the search tool, as in the agent-searched systems of §2

9.2 What would decide each claim

Each deciding comparison carries its own criterion, fixed before any run. A difference between two arms is a paired comparison on the same cases (McNemar, 1947) at $\alpha = 0.05$, sized for 80% power with up to a fifth of the cases discordant (Connor, 1987): about 155 cases detect a 10-point difference in hit at 5, and about 1,750 a 3-point one. A smaller gain than a test is sized for is reported with its 95% interval and not adopted. A bar of "every case" is met by zero failures in at least 300 cases, which bounds the failure rate below 1% at 95% confidence.

A "no worse" bar is a margin m: the arm passes when the 95% bound of the difference stays within it. That needs about $7.84\,\psi/m^2$ cases at 80% power, where ψ is the share of cases on which the two arms disagree. A 2-point margin at $\psi = 0.2$ needs about 3,900 cases, decades of corrections, so the owner's stream carries only margins it can resolve on 155 cases, each stated with the ψ it assumes, and tighter ones are checked on the synthetic suite, where cases cost nothing.

No question is decided on the owner's cases before the evaluation stream holds the cases its test needs; until then the synthetic suite (§9.4) carries it and the owner's cases are reported as a pilot. At three corrections a week, the rate in the example of §9.3, 155 cases take about a year, and longer as the design works, since each catch removes a correction. So the rate is counted in the session record before building, and the date of the first verdict printed. The arm comparisons on the owner's cases are then tested in an order fixed before any run, each at $\alpha = 0.05$, stopping at the first that fails: B against A, C1 against B, C2 against C1, E against the catalogue arm, E against pull only, fading. A fixed sequence keeps the chance of any false adoption at 5% without raising the 155 cases each test needs. The bars in Table 8 are adoption thresholds, not predictions.

9.3 Live measures

In use, four numbers tell whether the design works: the rate of the owner's "you have this" corrections, which should fall; the share of shown entries with a verified success, which should rise; the explicit-failure rate and the cache-write tokens before the memory block, which should not rise. The owner gets them as one line a week in the learning stage's morning report (J5 §7.5), for example "you told me 'you have this' three times this week; the items I missed were the lab-camera note and last month's training setup", so he can see whether the design works without opening the inspector.

Table 8. The questions the replay decides, and the bar each must clear.

Question	Decided by	Bar to adopt
Can a miss be placed at a gate?	Two raters assign each failed case its first failed gate, apart	Cohen's kappa at least 0.8 over at least 100 failed cases; at most one failed case in ten fits no gate
Does the asset index pay?	B against A, hit at 5	Higher, paired test on 155 cases; no more tokens pushed
Do use keys pay?	C1 against B	At least 10 points, on 155 cases; no more tokens
Do keys cut from corrections pay?	C2 against C1	Higher hit at 5 on 155 cases
Do co-access and the task-reference state pay?	D against C2, on synthetic cases sharing no word with the item	At least 3 points on 1,750 synthetic cases (a check of the mechanism, §9.4); on the owner's cases, reported with its interval
Is a pushed pointer used?	E, followed, among the cases hit at 5	Four in five followed before the owner is asked, with a lower 95% bound of at least two in three, on the cases hit at 5 among the 155; followed wrongly in at most one negative turn in twenty
Does it beat a strong asset-aware design?	E against the catalogue arm, at matched tokens and latency	Higher hit at 5 on 155 cases; explicit-failure rate at most 5 points higher (95% bound, $\psi \leq 0.05$), 2 points on the synthetic suite
Does the named slot work?	E on eligible named cases; the negative set	Every case, every time; named entries pushed on the negative set reported
Is access respected?	Items outside the reader's scope, quarantined, suppressed or erased, named and unnamed; derived items of sources with partly different readers; task references after access is lost, also mid-query; direct file reads	Zero entries, pointers or counts leaked in at least 300 cases per route; every direct read refused
Is push worth it at all?	E against pull only	Fewer misses on 155 cases; explicit-failure rate at most 5 points higher (95% bound, $\psi \leq 0.05$); cost within §7's bounds
Is placement free?	Cache writes per turn with and without the list, on the same replayed sessions	No extra rewrite of the unchanged prefix; added writes equal the list's own tokens within 5%; also across the night's publish of the index file (§4.3)
Is a correction retrievable at once?	Correction, then a relevant request in the same session, including one that matches only the old wording, and a repeat of the corrected request	Returned by a pull on the next request, pushed on a repeat of the request, old revision never served, every case; also across an index swap with a concurrent ordinary correction
Does "last time" resolve right?	The replay's cases with a mention that names nothing	A wrong reading in at most one resolved case in twenty; when two sessions fit, the agent asks
Does fading pay?	The best arm with and without fading	Higher hit at 5 on 155 cases; no eligible named or important case lost
Does the push fit the path?	Push latency, 95th percentile	50 ms or less
Does asking a session pay?	155 replayed "what did we decide about X" questions (the first 30 as a pilot), wake against search	Correct at most 10 points less often (95% bound, $\psi \leq 0.2$); fewer tokens where k^* predicts it; faster at the median

9.4 Public suites

LongMemEval (Wu et al., 2024) and LoCoMo (Maharana et al., 2024) test the text index against published systems. A synthetic ownership suite tests the asset index: generated catalogues of skills, machines and past approaches, and requests that need an asset while sharing no content word with it, split into development and held-out halves before any tuning. The generator also writes fetch logs, each task fetching a planted set of entries together with noise, and it writes both the "use when" text and the requests that must match it. So D against C2 and C1 against B on this suite check the mechanism, that the code finds what the generator planted; whether the keys are worth having is for the owner's replay. Some cases pair a request in one language with keys in another (§6.2). The suite runs from a clean copy and prints hit at 5, tokens pushed and the gate each miss failed.

9.5 Integration checks

At the start of a session, under each window controller, the test reads the model's tool list directly and looks for the pull and its range mode; a session without them fails. Hidden-row counts carry a reason, no probe counts as access, and the planted opportunities are found (Appendix C). Each handoff of this stage in the transition specification is a check (Table 10), and the series' design model is broken six ways to show that the checks can fail; all six are caught, and no implementation of Find has run. The seeded defects, two races kept as regression cases and three packing checks are listed in Appendix C.

10. Conclusions

Indexing what the owner owns by when it helps, pushing a few eligible pointers where they leave the cache intact, and letting the owner's replayed corrections decide what earns its place should cut the reminders, if the tests of §9.2 pass. On the morning of §1 (Table 2), the browser helper would arrive through its use key, the lab machine through its name, and "last time" through the task reference and a session card. If it holds, the assistant remembers what you built together without being told.

The design has limits. The owner's corrections are few and private, so the main test fills slowly, about a year to its first verdict at three a week, and the synthetic suite has to show that its results carry over. The replay sees only the misses he named, never those he worked around in silence. The push runs on the owner's words, on failed tool calls and on a draft reply that says it cannot; a need that appears mid-task with none of these waits for the agent to search. Use keys written by a model may be wrong, and pruning removes only the generic ones. Verified successes are rare, so the ranking leans on audited unchallenged uses. A pointer in front of the model is not yet a pointer used, and only the replay's *followed* measure can show the difference. The filters are only as good as the scopes and links they are given. An entry needed once a year ranks low on the day it is needed unless it is named or marked important.

11. Next steps

The instruments come first: the owner's-words field, the use log and receipts, planted opportunities and the pull check. Then the eligibility function, the asset index with the named slot, the overlay with its revision map and the replay sets; then use keys, co-access, task references and fading, each adopted only when it clears its bar in §9.2. The synthetic suite, which others can run, goes in parallel. The design will be built in TinkerClaw, an open-source fork of the OpenClaw agent framework, and tested as above, and an updated version will report the results.

References

Anderson, J. R., and Schooler, L. J. (1991). Reflections of the environment in memory. *Psychological Science*, 2(6), 396–408. https://doi.org/10.1111/j.1467-9280.1991.tb00174.x

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., and Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, 111(4), 1036–1060. https://doi.org/10.1037/0033-295X.111.4.1036

Anthropic (2025a). Equipping agents for the real world with Agent Skills. Anthropic Engineering. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills

Anthropic (2025b). Introducing advanced tool use on the Claude Developer Platform (Tool Search Tool), 24 November 2025. Anthropic Engineering. https://www.anthropic.com/engineering/advanced-tool-use

Anthropic (2026a). How Claude remembers your project (Claude Code memory). https://code.claude.com/docs/en/memory

Anthropic (2026b). Prompt caching. Claude Platform documentation. https://platform.claude.com/docs/en/build-with-claude/prompt-caching

Anthropic (2026c). Skill authoring best practices. Claude Platform documentation. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/best-practices

Anthropic (2026d). Mid-conversation system messages (turn-scoped messages, `clear_at`, beta). Claude Platform documentation. https://platform.claude.com/docs/en/build-with-claude/mid-conversation-system-messages

Anthropic (2026e). Using agent memory (Claude Managed Agents memory stores, beta). Claude Platform documentation. https://platform.claude.com/docs/en/managed-agents/memory

Anthropic (2026f). Use Claude's chat search and memory to build on previous context. Claude Help Center. https://support.claude.com/en/articles/11817273-use-claude-s-chat-search-and-memory-to-build-on-previous-context

Chhikara, P., Khant, D., Aryan, S., Singh, T., and Yadav, D. (2025). Mem0: Building production-ready AI agents with scalable long-term memory. arXiv:2504.19413. https://arxiv.org/abs/2504.19413

Chen, Y., Yoon, J., Sachan, D. S., Wang, Q., Cohen-Addad, V., Bateni, M., Lee, C.-Y., and Pfister, T. (2024). Re-Invoke: Tool invocation rewriting for zero-shot tool retrieval. *Findings of EMNLP 2024*. https://arxiv.org/abs/2408.01875

Church, K. W., and Hanks, P. (1990). Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1), 22–29. https://aclanthology.org/J90-1003/

Connor, R. J. (1987). Sample size for testing differences in proportions for the paired-sample design. *Biometrics*, 43(1), 207–211. https://doi.org/10.2307/2531961

Gan, T., and Sun, Q. (2025). RAG-MCP: Mitigating prompt bloat in LLM tool selection via retrieval-augmented generation. arXiv:2505.03275. https://arxiv.org/abs/2505.03275

Guo, W., Wang, D., Wang, Z., Liu, H., and Xu, Y. (2026). T-Mem: Memory that anticipates, not archives. arXiv:2606.15405. https://arxiv.org/abs/2606.15405

Gutiérrez, B. J., Shu, Y., Qi, W., Zhou, S., and Su, Y. (2025). From RAG to memory: Non-parametric continual learning for large language models (HippoRAG 2). *ICML 2025*. https://arxiv.org/abs/2502.14802

HippoRAG (2026). HippoRAG repository, README, "Upgrading". OSU NLP Group. https://github.com/OSU-NLP-Group/HippoRAG

LangChain (2026a). LangMem: hot path quickstart. https://langchain-ai.github.io/langmem/hot_path_quickstart/

LangChain (2026b). LangMem: conceptual guide. https://langchain-ai.github.io/langmem/concepts/conceptual_guide/

Letta (2026). Memory blocks. Letta documentation. https://docs.letta.com/guides/agents/memory-blocks

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. (2024). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173. https://arxiv.org/abs/2307.03172

Maharana, A., Lee, D.-H., Tulyakov, S., Bansal, M., Barbieri, F., and Fang, Y. (2024). Evaluating very long-term conversational memory of LLM agents (LoCoMo). *ACL 2024*. https://arxiv.org/abs/2402.17753

Marr, D. (1971). Simple memory: A theory for archicortex. *Philosophical Transactions of the Royal Society B*, 262(841), 23–81. https://doi.org/10.1098/rstb.1971.0078

McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2), 153–157. https://doi.org/10.1007/BF02295996

Mem0 (2026). Feedback mechanism. Mem0 platform documentation. https://docs.mem0.ai/platform/features/feedback-mechanism

Moon, S., Jha, S., Erdogan, L. E., Kim, S., Lim, W., Keutzer, K., and Gholami, A. (2024). Efficient and scalable estimation of tool representations in vector space (Tool2Vec). arXiv:2409.02141. https://arxiv.org/abs/2409.02141

Nogueira, R., Yang, W., Lin, J., and Cho, K. (2019). Document expansion by query prediction. arXiv:1904.08375. https://arxiv.org/abs/1904.08375

OpenAI (2026). Memory in ChatGPT. OpenAI Help Center. https://help.openai.com/en/articles/8590148-memory-in-chatgpt

OpenClaw (2026). Active Memory. OpenClaw documentation. https://docs.openclaw.ai/concepts/active-memory

Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., and Gonzalez, J. E. (2023). MemGPT: Towards LLMs as operating systems. arXiv:2310.08560. https://arxiv.org/abs/2310.08560

Rasmussen, P., Paliychuk, P., Beauvais, T., Ryan, J., and Chalef, D. (2025). Zep: A temporal knowledge graph architecture for agent memory. arXiv:2501.13956. https://arxiv.org/abs/2501.13956

Reimers, N., and Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *EMNLP-IJCNLP 2019*. https://arxiv.org/abs/1908.10084

Teyler, T. J., and DiScenna, P. (1986). The hippocampal memory indexing theory. *Behavioral Neuroscience*, 100(2), 147–154. https://doi.org/10.1037/0735-7044.100.2.147

Tulving, E., and Thomson, D. M. (1973). Encoding specificity and retrieval processes in episodic memory. *Psychological Review*, 80(5), 352–373. https://doi.org/10.1037/h0020071

Wang, Y., and Chen, X. (2025). MIRIX: Multi-agent memory system for LLM-based agents. arXiv:2507.07957. https://arxiv.org/abs/2507.07957

Wegner, D. M. (1987). Transactive memory: A contemporary analysis of the group mind. In B. Mullen and G. R. Goethals (eds.), *Theories of Group Behavior*, 185–208. Springer. https://doi.org/10.1007/978-1-4612-4634-3_9

Wu, D., Wang, H., Yu, W., Zhang, Y., Chang, K.-W., and Yu, D. (2024). LongMemEval: Benchmarking chat assistants on long-term interactive memory. arXiv:2410.10813. https://arxiv.org/abs/2410.10813

Zep (2026). Graph namespacing. Graphiti documentation. https://help.getzep.com/graphiti/core-concepts/graph-namespacing

Zou, M., Guo, Z., Liang, L., Wang, Z., Wang, Q., Wen, Q., King, I., Qu, L., and Xu, Z. (2026). Remember the decision, not the description: A rate-distortion framework for agent memory. arXiv:2605.10870. https://arxiv.org/abs/2605.10870

Appendix A. Interfaces

These are the records Find writes and reads. Field names are illustrative; the fields are the contract.

Table 9. The records of the finding stage.

Record	Written	Fields
Push or pull receipt	every turn, every failed tool call or draft reply that triggers a push, and every pull, even when empty	turn ID; trigger (owner message, failed tool call or a draft that says it cannot); how a "last time" was read, if any; reader scope and permission version; index build ID and age; snapshot and erasure epoch served; mode (full or lexical only); candidates per key type; rows removed by each filter step (audit permission only); entries shown as ID@revision with form, reason and p_e; latency
Use-log row	per entry shown	turn ID; entry ID@revision; route (push, named slot, pull); shown, with its certainty (known input, or sent but unconfirmed); used; body read; outcome (verified success, explicit failure, unchallenged use, unknown); missed by push; the key that matched
Index manifest	every build	index ID; schema version; encoder and version; tokenizer; s_0 and s_1 per key type and scorer; snapshot read and its erasure epoch; counts of entries, keys and edges; result of each publication and swap check
Overlay receipt	every accepted write, hold or pending line	fact or rule ID@revision, or hold; J14's receipt_id as key, closing the delivery record via `acknowledged_by` (J14 A.2, J14's Table 6); snapshot (none for a hold or pending line); applied epoch; time; status (retrievable, served as a hold, or queued with its retry count)
Revision map entry	every changed or erased ID since the build, kept across a swap while newer than the build's snapshot	ID; latest revision or tombstone; write or deletion receipt ID
Use key	at write, from a correction, or at night	text; for a key cut from a correction, the request it came from, as a second key text; entry ID; dependency links to the addresses it came from; reader scope: the readers all of them admit
Task-reference state	end of every turn	session ID; task title and where it came from (first request or session card); up to twenty entry IDs; unresolved references

The four stages share a transition specification (J5 Appendix B): a model of the design, replayed with a crash, a duplicate and a reordering at every handoff, that gives each handoff a code. Table 10 lists the ones that reach this stage, with the test that checks each.

Table 10. This stage's handoffs in the transition specification.

Code	What arrives or leaves	Where	Test
T4	a write decided while the writing stage's judge is down	§8.2	the pull shows it as unchecked, the push never shows it
T9	the delivery slip for an accepted write	§8.2, App. C	a query on the old wording misses the old revision
T10	Find's confirmation that part of a correction is retrievable	§8.2	the tail line leaves only when every part is retrievable or held
T12	a held write	§8.2	served only as a hold, only to the readers §8.1 allows
T13	a night build starts	§8.3	its manifest records snapshot, epoch and anchors
T15, T16	an eviction event after a compaction or a rebuilt window	§6.7	the ledger never answers from an evicted range
T17	an erase order	§8.3, App. C	an older build is refused; Find's deletion row only after the refusal
T19	a night build finishes	§8.3	a correction made during the build is found after the swap
T20	a narrower reader names an item or reads a file	§8.1, §6.4	nothing, not even a count; the file read is refused
T21	the owner narrows a reader's scope	§8.1	a query in flight is checked again when it returns
T23	a quarantine order	§8.1	only an audit view shows the rows

The body cites the companion papers by section only. Table 11 says, once, what this stage takes from each.

Table 11. Where the companion papers specify what this stage relies on.

Section	What this stage takes from it
J1 §4.1	the owner's words, kept apart from the wrapper (§6.1)
J1 §4.7	sessions that are never kept (§4.1)
J1 §6.1	the three window controllers (§6.6)
J1 §6.6	eviction events, and how sure delivery is (§6.7, §6.8)
J1 §7.2	the price of moving the prefix (§7.4, Table 5)
J1 §8.1, §8.3	the range read; asking a past session (§6.6, §6.9)
J14 §4	choosing what to write (Table 5)
J14 §5.2, §5.3	every writer through one gate; the pending area while the judge is down (§4.3, §8.2)
J14 §7.2	delivery slips, epochs and concurrent writers (§8.2, §8.3, Appendix C)
J14 §7.3, §7.4	three ways a fact ends; two questions about the past (§8.1)
J14 §7.5, §7.7	only a judge closes; erase and quarantine (§3, §8.1)
J14 §8.1, §8.5	source classes and links; dates, confirmation and importance (§5.5, §8.1)
J14 §9.2, §9.3, §9.4	one topic per file; pointers only in the index file; use keys written with the fact (§4.3, §5.3, §6.7)
J5 §4.1	the rule ledger, one row per rule (§8.2)
J5 §6.1, §6.2	the fast path for corrections; the nightly auditor (§6.5, §6.9, §8)
J5 §7.1, §7.2	the index file's budget; prune by binding (§4.2, §4.3)
J5 §7.4, §7.5	approval and who may loosen; the morning report (§4.3, §8.5, §9.3)
J5 §9.1, §9.2	a receipt for every run; three streams kept apart (§8.4, §8.5)
J5 Appendix B	the transition specification and the reference trace (§2, Table 10)

Appendix B. The push in pseudocode

```
FIND_PUSH(turn, reader, query=None, rerun=False):
  # reader: authenticated state. Budget 50 ms (p95):
  #   words, task refs 2; epoch fence <=10; encoder 15;
  #   key match 15; eligibility, scoring 6; packing 2
  e0 = newest_epoch
  wait = 0 ms if rerun else 10 ms               # §8.3 fence, bounded
  if not applied_within(e0, wait):
      return empty(turn, "skipped: erasure in progress")
  words = query or owner_words(turn)            # §6.1
  mode  = FULL
  v0    = permission_version()
  view  = authorised_view(reader)               # §8.1 step 1
  named = view.names.match(words)               # §6.4
  named += task_ref_state.resolve(words)        # §6.1, "last time"
  vec = with_timeout(15 ms, embed, words)       # §6.2
  if vec and manifest.ok(encoder):
      cand = view.keys.match(vec) + view.bm25(words)   # hybrid
  else:
      cand = view.bm25(words); mode = LEXICAL
  cand += view.overlay.match(words)             # §8.2
  cand += side_search_hits(previous_turn)       # §6.2, marked
  cand  = eligible(cand, reader)                # every route
  cand += eligible(one_hop(cand, view.coaccess), reader)
  named = eligible(named, reader)
  for e in cand: e.p = logistic(score(e))       # §5.5
  ranked = top([e in cand - named if e.p >= TAU], 5)
  if newest_epoch != e0:                        # an erasure arrived
      if rerun: return empty(turn, "skipped: erasure in progress")
      return FIND_PUSH(turn, reader, query, rerun=True)  # once
  if permission_version() != v0:                # checked again
      named  = eligible(named, reader)
      ranked = eligible(ranked, reader)
  extra = named[3:]; named = named[:3]          # past three: one line
  pack  = named + names_line(extra) + ranked    # may be empty
  for e in pack:                                # §7.3, cap §6.3
      whole = e.tokens < B_star(q(e)) and fits(pack, 1000)
      e.form = WHOLE if whole else POINTER
  place pack in turn-scoped slot if offered,    # §6.5
        else after the cached prefix, or after the failed result
  for e in pack: use_log.shown(turn, e)         # §6.8
  emit receipt(turn, pack, counts, mode, snapshot, epoch)

ON_TOOL_FAILURE(call, turn, reader):            # §6.1
  failed = call.error or not call.status.ok \
           or page_kind(call) in (LOGIN, CAPTCHA)
  if failed:
      q = [call.tool, call.args, failure_kind(call)]
      FIND_PUSH(turn, reader, query=q)          # never result text
```

```
ON_DRAFT_REPLY(draft, turn, reader):            # §6.1, once a turn
  kind = claim_kind(draft)                      # "I can't", "which?"
  if kind:
      FIND_PUSH(turn, reader, query=[kind, owner_words(turn)])
      redraft(turn)                             # query: no draft text

eligible(xs, reader, mode=CURRENT):   # every route, task refs too
  xs = resolve(revision_map, xs, mode)
      # the one revision each would return; drop or swap only
  xs = [x for x in xs if authorised(reader, x)]
      # §8.1 step 1, on that exact revision
  xs = drop_quarantined_suppressed(xs)          # §8.1 step 2
  return visible(xs, mode)                      # §8.1 step 3
```

Appendix C. Freshness and integration checks in detail

Delivering a write. In the same transaction as the write, the writing stage sends a delivery slip (IDs, revisions and receipt IDs, never content) and resends it until Find confirms (J14 §7.2). Find fetches each published revision from the store's published snapshot, so a revision erased meanwhile arrives as a tombstone. A slip older than the latest erasure Find has applied is re-read from the store, and an older revision never replaces a tombstone or a newer one. A hold or a pending line has no snapshot: Find reads the record its write receipt names, at the epoch it was applied under. Find's confirmation carries that receipt ID and closes the slip through its `acknowledged_by` field (J14 A.2). From then on the item is retrievable, and the learning stage marks that part of the correction retrievable. *Test:* a slip delivered late, after an erasure of the same fact, never brings the erased revision back.

Restarting. After a restart, the overlay and the revision map are replayed from the writing stage's receipts since the manifest's snapshot, so a restart loses no accepted write. A change of encoder is a rebuild; a change of chat model is not.

Swapping a build. A build reads the store up to one snapshot, its watermark. The swap installs the new index together with the overlay rows and revision-map entries newer than that, and drops only what the build already holds. *Test:* a correction committed at snapshot 104 while a build from snapshot 103 runs, with the epoch unchanged, is found after the swap without another rebuild. If the epoch moved, the build is refused, the live index and overlay keep serving, and Find fills its row of the deletion receipt only after the refusal, since until then the old build could still swap in.

Reading across an erasure. Find answers only after applying the newest erasure or quarantine epoch: an earlier read waits, and a read started under an older epoch that returns after a newer one is discarded and run again, so no query in flight serves the older state. The push must not stall a turn while it waits. It waits at most 10 ms of its budget; past that, it sends an empty list with a receipt reading "skipped: erasure in progress" and leaves the finding to the pull, which may wait. A push caught across an epoch change runs again once, without waiting at the fence, and sends the empty list if the epoch is still moving (Appendix B). *Test:* during a large erasure every turn still gets its receipt within the 50 ms budget at the 95th percentile, and none serves an erased row.

Watching receipts (§8.4). Three things are watched apart. *Liveness*: every turn produces a push receipt, empty or not. *Opportunity*: a planted request, run under a test identity, must find its known asset. *Outcome*: entries shown and used, outcome labels, pulls, read-ledger hits and rows removed per filter are reported, never alarmed at zero.

Table 12. Moving into "retrievable": failure cases and what may be said. The whole journey: J5 Appendix B, the reference trace.

Case	What Find does	Status the agent may name
Overlay append fails, delivery lost, or the writing stage crashes after publishing	resent from the writing stage's outbox until acknowledged	accepted, not yet retrievable
Writing stage's judge down	line kept in its pending area; the pull shows it as unchecked, the push never	captured, acceptance pending
Write held	served only as a hold	held for the owner's approval
Restart before the night	overlay and revision map replayed from the receipts since the build's snapshot	retrievable once the replay ends
Concurrent writers	the revision that won the commit wins here too (J14 §7.2)	retrievable at that revision
Re-index or new encoder	new index built beside the old; swapped in with the newer overlay suffix (§8.3)	retrievable throughout
Compaction of the session	read ledger drops the evicted ranges (§6.7)	retrievable
Privacy erasure	overlay rows, use keys and ledger range dropped; the ID tombstoned in the revision map; read service waits for the new epoch and reruns a read in flight; older build refused (§8.3)	erase status, per the writing stage
Reader outside the scope, or scope narrowed during a query	nothing shown, not even a pointer or a count; `eligible()` runs again at return	not retrievable for this reader

Integration checks (§9.5). The series' design model is broken six ways, one at a time: the swap empties the overlay; access is checked at collection only; "last time" references skip that check; reads ignore an erasure in progress; a derived item takes the readers of its most restricted source, not those all its sources admit; a hold takes its topic's readers. All six are caught. Two races stay as regression cases, because a query that skips the check at return passes them silently: an erasure mid-query hands the owner an erased phone number, and a quarantine mid-query a use key made from injected text. With the check, both rerun and serve nothing forbidden. Three packing checks join them: an oversized top hit does not hide smaller hits, a new session's first push draws on the latest entries of the owner's previous session, and the full-text query never exceeds 2,000 characters.

Appendix D. The owner's inspector

The owner's inspector is one owner-only view over the use log and the receipts (§6.8). For any item it shows the sources (addresses) and the state in both vocabularies: J14's receipt word (accepted, pending, held, refused), with the receipt and its attempt and previous_receipt chain,

and this stage's status from Table 12, such as retrievable or not retrievable for this reader; a pending or held line never shows as a fact. It keeps three actions apart: correct (a new write through the writing stage's gate), don't mention (the suppression above) and delete (J14 §7.7, whose links list the derived places a delete must reach). ChatGPT's help page likewise separates saved memories from chat history and says deleting a memory leaves the earlier chats that mention it (OpenAI, 2026); the inspector lists such leftovers. *Test:* each state of Table 12 shows under its own label, and a held item never shows as current.

J-series paper J2 | Version 8.7 | 2026-10-21