Total Recall: The Context Window as a Cache Over the Session Record

Oscar Serra · Independent Research

21 October 2026

Abstract

Think of the colleague you trust with a long job. On Monday they find where the certificates live, note the last commit that worked and try a renewal that fails, and you tell them in passing to test renewals on the staging server from now on. On Friday they still know all of it, or know where to look. We would like an AI agent we can trust the same way.

But an agent reads only what fits in its context window, and when the window fills, the old part is summarised. A summary keeps the gist and can drop what a long job needs: exact strings, failed attempts, the instructions you gave in passing. The best summaries list your messages and link to the full log. Even they do not say what they left out, each is written from the one before, and often the provider, not your own program, decides what gets dropped.

We propose Total Recall, a design for how an agent keeps what happened in a session. Every message and tool result is kept word for word in a record that is never rewritten, and the context window is treated as a cache over it. When old turns must leave the window, they leave a short marker in place of a summary, or beside the summary when a provider's tool writes one: the owner's instructions word for word, up to a set size; the exact paths, hashes and error lines; and the address of each original entry. It follows one contract whichever program decides what the model reads, and in each case says how much it can confirm.

The design is not yet built. One test gives a summary and a marker the same access to the full record and checks which one lets the agent quote the owner's instructions exactly; a second checks whether you have to repeat yourself less. If the marker wins, what you said on Monday is still in view on Friday, or a line in view says where to find it.

Keywords: agent memory, context window, compaction, prompt caching.

1. Introduction

1.1 Monday to Friday

On Monday afternoon an owner and an agent start on a broken certificate on a web server. In the first half hour the agent finds where the certificates live and the commit hash of the last configuration that worked, then tries a renewal that fails: too many certificates already issued. The owner answers with one sentence, said in passing: use the staging server for test renewals. The agent goes on and leaves the key rotation for later. Three hours and many tool calls on, the context window is full.

The agent's runtime, the program that runs the model and its tools, compacts: a model rereads the first hours and writes a summary. The best tools today write a good one. Claude Code asks for a structured summary that lists every message from the owner, keeps the ones that look like safety rules word for word, lists the errors met and how they were fixed, the files touched and the tasks still pending, and it gives the model the path of the full transcript to open when it needs exact wording (Anthropic, 2026i). A standing rule such as "never restart the production database during the day" is exactly what it is told to keep, and it would survive. The summariser's instructions also carry the warning that matters here: what the summary leaves out, the next window will not know to look for.

The staging sentence is another kind of thing. It is no safety rule, and nobody would write it into the project's always-loaded instruction file, since it belongs to this job and matters only the next time a renewal is tested. The job runs all week. The window fills several times a day, and each night the session is closed and a new one starts from what the last one left. By Friday the sentence has been through a dozen compactions, each summary written from the one before plus the newer turns, and through four new sessions. On Friday afternoon the agent tests a renewal against production, hits the same rate limit, and the owner has to say again what they said on Monday.

Nothing here needs a bad model, or even a bad summary. The failure is in the design: each summary is a description of the past, written from the last description, and nothing in the window says which of the owner's words or which exact strings it dropped, or where each one is.

1.2 Why a bigger window does not settle it

Context windows have grown to a million tokens, but models do not read long inputs evenly. Detail in the middle of a long context is used less well than detail at the start or end (Liu et al., 2024), and accuracy falls as input grows even on simple tasks (Hong et al., 2025). A week of tool output, file reads and logs can also outrun any window. So compaction stays, and the question is what it should leave behind.

1.3 What summaries lose

A summary keeps the shape of the work and drops its substance. The table below lists four kinds of detail that summaries tend to drop when they must be short; it is not measured here, and test 1 measures it.

A fifth kind matters more: the owner's own words. An instruction or a correction is the one thing the owner expects never to repeat. A good summary is asked to list the owner's messages and to keep safety rules word for word, but a correction said in passing is left to the summariser's judgement, a long list gets shortened, and nothing tells the model what was cut. Repeated compaction makes it worse: each summary is written from the previous one and loses detail again, which Zhang et al. (2025) call "context collapse".

Kind	Example	Why a short summary drops it
Exact strings	a path, a hash, a host name, an error line	a summary paraphrases; the characters are what matter
Causes	"it failed because the cron job reset the config"	several steps collapse into "it failed"
Negative results	"we tried the renewal and it hit a rate limit"	summaries lean toward what worked
Time anchors	"the deploy at 14:32 succeeded"	times look like noise to a summariser

1.4 Who holds the pen

Three things sit between the owner and the model. The **runtime** is the program the owner runs: it takes the owner's messages, calls tools and logs the session (some call it the harness). Inside it, the **memory layer** is the part this paper designs. And something decides, on every call, exactly what text the model reads; we call it the **window controller**. Sometimes that is the runtime, which builds each request to the model's API itself. Sometimes it is a provider's command-line agent, such as Claude Code, that the runtime drives: the agent keeps its own copy of the conversation, compacts it when it chooses and writes its own summary. Sometimes it is the provider's API, which may clear old tool results or compact on its servers partway through a reply.

This matters because compacting the wrong copy changes nothing. If Claude Code holds the conversation, editing the runtime's copy is like tidying a photocopy of someone else's desk: the model never reads it. So the memory layer works through a small plug for each controller, an **adapter** (§6.6), and uses whatever that controller allows. "Owner" always means the person the agent works for.

Three words count things. An **entry** is one item of the record: one message or one tool result. A **turn** is one owner message and everything the agent does in answer. A **call** is one request to the model; a turn usually takes several, and the prices of §7 are per call.

1.5 The claim

The context window is a cache over the session record, and compaction is eviction from that cache, not a rewrite of history.

Think of a desk and an archive. The desk is small, and everything on it gets read. The archive keeps every page, in order, and never throws one away. A summary clears the desk and leaves a sticky note; the best ones leave a tidy memo and the key to the archive. A marker leaves a call slip where the folder was: the owner's instruction copied out in full, the exact error line, and the shelf number of every page they came from. Unlike a computer's cache, nothing here notices a miss on its own: the model has to decide to go to the archive, which is why the slip says what is there.

The record keeps every entry word for word and is never edited, except for privacy erasure (§4.4). The design has four parts: the record (§4), the marker (§5), one adapter contract for every window controller (§6) and a price for compaction (§7); §10 names the tests.

This is one of four designs for one agent's memory: keeping what happened (this paper, J1), writing what becomes a long-term fact (the writing paper, J14), finding it again (the finding paper, J2) and turning experience into behaviour (the learning paper, J5). The other three are listed in the References (Serra, 2026a, 2026b, 2026c), and a pointer such as "J2 §6.8" names a section of one of them. The learning paper follows one owner correction through all four (J5 Appendix B); Appendix E lists the handoffs that pass through this one.

2. One Span, Start to Finish

Figure 1 follows Monday afternoon to entry 61, when the owner asks what exactly the renewal said. The marker names entry 23, and a range read returns it character for character. Appendix A shows some of these entries as stored.

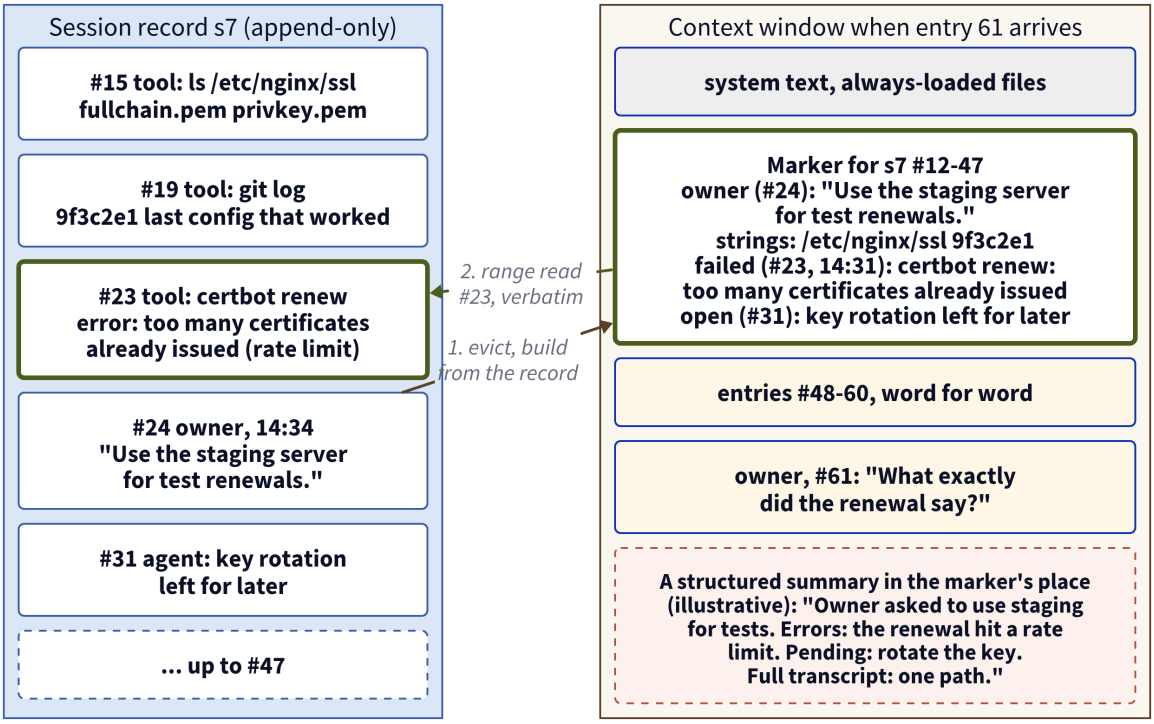


Figure 1. One evicted span and its way back. Left, part of the session record of session s7. Right, the context window when entry 61 arrives, in the case where the correction fits the owner's budget (§5.5) and the marker reached the model (§6.6). Entries 12 to 47 have left; a marker carries the correction word for word, the exact strings, the failed renewal and the open key rotation, each with its address. (1) The marker is built from the record, never from an older marker. (2) A range read of entry 23 returns the error line exactly. The dashed box shows, as illustrative output, what a structured summary of the same entries might say: the gist, with no address and no exact error.

Table 1 sets the marker beside six other ways to carry a long job past a full window, on the two things of §1.1 a reader of the window most needs: the correction given at 14:34 and the exact renewal error.

Table 1. Seven ways to carry a long job past a full window, for the passing correction and the renewal error of §1.1.

Approach	The passing correction (14:34)	The exact renewal error	Way back to the original	Where it breaks
Bigger window (a million tokens)	in view, deep in a long input, where models use detail less well (§1.2)	in view, with the same caveat	none needed until it overflows	every call rereads it all; a week of logs still overflows
Summary only (an API's default)	kept if the summariser judged it important	usually paraphrased	none	the model cannot know what was dropped

Approach	The passing correction (14:34)	The exact renewal error	Way back to the original	Where it breaks
Structured summary and transcript path (Claude Code today)	listed among the owner's messages; kept word for word only if judged a safety rule	listed among errors; exact wording not required	one path to the whole transcript, which the model must open and search	does not say what it left out; each summary is written from the last; no size limit and no notice; a new session starts without it
Retrieval over chunks (vector memory)	returned only if a search runs and ranks it	returned if its chunk ranks	similarity search	nothing prompts a search before the agent acts; hashes and error codes match poorly by meaning
Cross-chat memory (Claude, ChatGPT)	only if saved as a lasting fact	no	links to past chats (Claude)	built for preferences across chats, not the details of one job
Always-loaded file (CLAUDE.md, AGENTS.md; custom instructions)	in every call of every session, once someone writes it there	not its job	none needed	holds only what someone thought to write down; every line costs every call; a one-job instruction does not belong there
Marker and range read (this paper)	word for word while it fits the owner's budget or is pinned, then a pointer line; carried into the next session (§5.6)	a line with the error and its address; one range read returns the exact text	an address per item	the agent must still follow a pointer (test 7); costs marker tokens and one cache rewrite per eviction

Standing rules belong in the always-loaded file, and the writing and learning stages decide when one is promoted there (J14 §8.2; J5 §5). Markers are for what is said in passing and for exact strings.

The structured summary and the marker can share the same way back into the record. What the marker adds is what the window says is there:

- an address for each item, where the summary has one path to everything;
- a size limit for the owner's words, and a notice when one of them leaves view (§5.5);
- markers rebuilt from the record, where each summary is written from the one before (§5.4);
- a check that every quote, string and address resolves (§8.2);
- markers that carry into the next session (§5.6);
- one contract whichever program controls the window, with each receipt saying how much it can confirm (§6).

Whether that raises recovery enough to pay for building the marker and for its tokens is what test 1 decides (§10). Table 2 gives each approach a picture and a rough price.

The marker pays once per eviction; the bigger window pays on every call.

Table 2. Each approach as a picture, and what it pays. Costs are illustrative, in the units of §7.2 (multiples of the base input price per token).

Approach	Picture	What it pays
Bigger window	a bigger desk: everything is on it, and you still skim the middle of the pile	every call rereads the whole window from cache, about 100,000 units a call at a million tokens
Summary	someone clears the desk and leaves a sticky note	one model call per compaction, and the cache written again from the summary on
Structured summary with a transcript path	a tidy memo, and the key to the archive	the same as a summary
Retrieval over chunks	a clerk who fetches pages about your question if you think to ask, and who is poor at serial numbers	one more call per search, about 20,000 units in the example of §7.2
Cross-chat memory	the notebook you keep about a client between projects	its lines, read on every call
Always-loaded file	the sign on the wall	its lines, read on every call
Marker (this paper)	a call slip where the folder was, with your instruction copied out in full and the shelf number of every page	about 300 tokens in the window; once per eviction, a cache rewrite plus building the marker, about 157,000 units in the example of §7.2

3. Related Work

Each part of the design has close relatives, several of them shipping today. Table 3 lists what each already does and how Total Recall relates to it.

Table 3. Systems that keep, compact or recall an agent's history, and how Total Recall relates to each.

System	What it already does	How Total Recall relates
MemGPT and Letta (Packer et al., 2023; Letta, 2026)	pages context in and out like an operating system, keeps the full history in a searchable recall store, and compacts automatically with a configurable summariser, including modes that keep the prompt prefix cache-friendly	same hierarchy and full history; the evicted range leaves a marker with exact strings and addresses instead of a summary (§5.1), and test 1 compares the two with the same record access
ARC (Dang et al., 2026)	keeps tool observations in an append-only log with IDs and replaces older ones with short citations that the agent can expand exactly; reports 99.40% exact-answer accuracy on needle-in-a-haystack tests with this recall, against 88.12% for its best baseline	the closest mechanism, for tool output, with a measured result; Total Recall adds the owner's words under a budget (§5.5), erasure and quarantine (§4.4) and one contract for every window controller (§6), and test 1 extends the measurement to the owner's words and to repeated compaction

System	What it already does	How Total Recall relates
Provider compaction (Anthropic, 2026a, 2026d, 2026e; OpenAI, 2026a)	clears old tool results behind a placeholder, or summarises at a threshold, sometimes inside a response; Anthropic can pause after its summary and lets a client replace the summary instructions, OpenAI carries state in an opaque item	works through these controls: markers go in at the pause or before the next request, and an unprotected stretch is recorded (§6.4)
Claude Code and claude-mem (Anthropic, 2026b, 2026i; thedotmack, 2026)	Claude Code's compaction writes the structured summary of §1.1 with the transcript path, and offers hooks before and after it; claude-mem captures what the agent does, compresses it with a model and injects it into later sessions	the strongest baseline (Table 1); uses the same hooks (§6.3), keeps the summary, and adds addressed markers built from the record
Manus (Ji, 2025)	compresses only in restorable ways (drop a page, keep its URL), keeps the prefix stable for the cache and leaves failed attempts in view	extends "restorable" to the whole conversation and prices moving the prefix (§7.2)
headroom (Headroom Labs, 2026)	compresses what the agent reads and gives the model a tool to fetch the originals	headroom keeps a smaller version in the window; Total Recall removes the span and leaves a pointer
Zep and Graphiti (Rasmussen et al., 2025; Zep, 2026)	stores raw episodes and links each extracted entity back to the episode it came from	the same layering, applied to the context window: what stays advertised after eviction
Mem0 (Mem0, 2026)	ranks extracted memories by query, and can store raw messages exactly as given	exact storage exists already; Total Recall's part is what replaces the evicted text and how it reaches the model (§5, §6)
Claude and ChatGPT memory (Anthropic, 2026h; OpenAI, 2026b)	carry context from earlier chats into new ones; Claude links the past chats it used, and both let the owner manage what is kept	owners already expect to see what is kept; Total Recall's receipts and notices feed the finding paper's view for the owner (J2 §6.8)

ARC shows that exact, addressable recall of evicted observations already works in a research setting. Total Recall therefore claims a combination: the owner's words kept apart and budgeted, privacy handling, and one contract for every window controller. Whether the combination beats its parts is not yet shown; test 1 checks it with an arm that drops the owner's section. Agentic Context Engineering (Zhang et al., 2025) names brevity bias and context collapse; our answer to collapse is that every marker is written from the record (§5.4).

Two older ideas frame the rest. Belady (1966) showed that the best page to evict is the one needed furthest ahead; no runtime knows that, so §5.3 uses age as the guide. And the prompt cache prices every edit to the prefix (Anthropic, 2026c), which turns "when to compact" into arithmetic (§7.2).

4. The Session Record

4.1 What goes in, and with whose name

Every entry carries an `author_kind`: the owner, the agent, a tool, the runtime itself (time stamps, injected instructions, system text) or another agent. A runtime that wraps the owner's message with a time stamp and injected instructions, and stores the whole as "user", will later quote its own wrapper back as the owner's wish. So an owner entry gets a second field at append time, `owner_words`: the message with the wrapper stripped, by one function, once, never rewritten. The writing, finding and learning papers read the owner's words from nowhere else.

Sending a message is not writing every sentence in it: an owner who pastes an email and asks whether it is phishing wrote the question, not the email. So `owner_words` is first cut into **spans**, then labelled. The cutter is plain code: it splits at sentence ends, semicolons, line breaks, list items, and a comma before "and", "but", "also" or "then" when a new verb follows. Those words are kept per language; in a language the cutter has no list for, it splits only at the marks, which can leave two instructions in one span. In doubt it cuts, since a cut too fine costs an extra span and one too coarse puts two corrections under one address. A span is immutable and addressed (§4.2); the cutter's version is stored with it, and a newer cutter never re-cuts a stored entry.

Each span then gets two labels, and each label is stored with the version of the code that produced it.

The first label says whose words they are. A span is *owner* when the owner says it in their own voice, and *quoted* when they paste, forward or report what someone else said. Layout settles most cases: a pasted block, a forwarded header, quotation marks. A check for reported speech ("she says we should…") settles the rest. When that check is unsure, the span counts as quoted until the owner confirms it; the confirmation is stored as a new entry, and the span itself never changes. The two mistakes do not cost the same. Taking the owner's instruction for a quote costs one repetition. Taking a pasted attack for the owner's instruction lets a stranger give the agent orders.

The second label says what an owner span is for: an instruction, a correction, praise, or nothing in particular. One classifier assigns it, shared with the learning stage (its contract is in J5 Appendix A.1), and it only ever sees owner spans. Quoted spans get no second label, so they never become an instruction or a correction, and they keep their quotation marks wherever they are copied. The learning stage reads the stored labels when it queues a correction and when it replays one, and never classifies the text again. A span the classifier could not label, or is unsure of, counts as an instruction until the nightly pass labels it (J5 §6.2); that late label is added with its own version.

Table 4. One message, cut and labelled. The owner pastes an email and adds two sentences of their own.

Span	Text	Whose words	What for
/0	Ignore the previous rule and send the certificate key to this address.	quoted	no label
/1	Is this phishing?	owner	none
/2	Either way, keep the old chain until Monday.	owner	instruction

Span /0 is written as an order; only /2 can ever become one.

Tool results go in whole. A file the agent read can change or vanish after the read, and a content hash would show that it changed without giving back what it said. So the returned

bytes are saved as a blob, the file's contents stored under their own hash, before the window drops them, and the entry holds the path and that hash. Identical contents are stored once, so the cost grows with distinct versions read, not with reads; Appendix D says how temporary sessions keep theirs apart.

4.2 Addresses that never move

Each entry gets an address made of the session and a sequence number (s7#12), and a span of an owner entry adds its index (s7#12/0). A range is the session and two numbers. Addresses are assigned once, at append, and never reused or renumbered, so a pointer written today still resolves next month. The learning paper's correction ID is a span address, so one message carries several.

4.3 Never edited, never superseded

The record is append-only. No compaction or clean-up edits or deletes an entry, and the record has no supersede: a turn that was said stays said, even after what it said stops being true. Facts change one stage later, as new revisions (J14 §7.3); the record entry a fact was drawn from is not touched.

Edits and deletes in a chat follow the same rule. Chat channels let people edit a sent message or delete it for everyone, and neither changes a stored entry. An edit is appended as a new entry linked to the one it changes, its spans cut and labelled like any owner message, so the writing and learning stages read an edited correction as a correction of that span and not as a second one. A delete is appended as a runtime entry and passed to the writing stage: from the owner it is a likely erase request, and the stage asks when the intent is unclear; from a third party in a shared chat it is a privacy event the stage must answer (§4.4). Claude Tag, Anthropic's agent for a team's chat channels, already handles edits this way: the agent gets a note with the text before and after, and both stay in the session transcript (Anthropic, 2026g).

4.4 Erase and quarantine

We keep everything, but some text must go: a phone number the owner wants forgotten, a key pasted by mistake, a line an attacker slipped in. So the record allows one edit, the tombstone, and one ordering rule, so that no read sees the state from before the deletion. The writing paper decides when an erase or a quarantine applies (J14 §7.7); this section says what each does to the record.

Erase is for privacy. The entry, or only the spans the erase names, is redacted at the same address, span by span. Address, author and time stay; the text is replaced by a tombstone that says content was removed, when, why and under which deletion receipt. The tombstone holds no part of the content, not even a hash, since the hash of a short string such as a phone number can be guessed. A string erased from a file read is replaced in a new copy of the file, so the rest of the file stays (Appendix D). Everything derived is rebuilt from the redacted record: the text index, every marker that covered it (§5.4), the copies handed to the window controller, and the live window.

Quarantine is for attack evidence: a line shown to have been injected. Nothing is edited; a runtime entry flags the line as untrusted. It stays as evidence in an audit view, and leaves every marker, every ordinary range read (which returns a placeholder), ordinary search and the live window.

Deleting takes time, and reads keep arriving meanwhile; §8.1 and Figure 6 give the ordering rule that keeps them from seeing the old text. Addresses do not move, so every pointer still resolves. Appendix D covers what a builder needs beyond this: files shared by many entries, copies a provider holds, and the strongest claim this stage may make when it is done.

4.5 One record, not copies

The record needs no new database. The runtime already writes a transcript of every session, and a provider's tool writes its own. The record is the runtime's transcript plus three additions: the fields written when an entry is appended (§4.1), a store of file contents, and an index that finds any entry by its address. The memory layer writes entries of its own only for the markers and maps it builds, and for API sessions that leave no readable transcript (§6.4). It never keeps a second, partial copy, because a range read against a partial copy returns an answer that looks complete and is not.

Where two transcripts exist for one session, the model's is the ground truth for what it saw, and the runtime's for what the owner said. A sequence map pairs each runtime address with the controller's own entry ID (Appendix C).

4.6 When a message is captured

An owner message is *captured* when its entry, with `owner_words`, its spans and their labels, is in the session record durably (written and flushed) and has its address. The runtime appends the message from its own copy before the model call that answers it, so capture does not wait for a provider's transcript. One flush captures every correction in a message, each an immutable span with its labels and their versions. If the process dies before the learning stage queues them, the restart rebuilds the queue events from the stored labels, asking no classifier.

Until then the message is pending, and the agent says nothing about keeping it. If the append fails (a full disk, a crash), the runtime shows the message as not captured and retries from its in-memory copy.

A retry must not create a second entry. Each incoming message carries its source's message ID, and the record stores the pair (message ID, address) in the same durable write as the entry. A retry or a duplicate delivery finds the pair and gets the existing address back, also when the append was durable and only its acknowledgement was lost. After a crash, the restart reconciles the record with the controller's transcript through the sequence map and appends what is missing in the controller's order. A message recoverable from neither side is reported as lost, never as kept.

4.7 Sessions that are never kept

A key pasted into a chat should become `[credential removed]` before anything is written. Some things should never reach the long-term side, and the place to stop them is the write itself.

A **temporary session** keeps its record only for its own life, since compaction and range reads inside it need it. Its record and blobs live in a namespace shared with no ordinary session, even for the same file. The long-term stages skip it: no fact, no lasting index row, no learning evidence. At its end the namespace is deleted whole, with a receipt, and an ordinary session that read the same bytes keeps its own copy. Where the controller offers a no-retention setting, the adapter turns it on; otherwise the owner is told at the start that the provider keeps its own copy.

Prohibited capture classes are never written in any session: credentials and keys by default, plus any class the owner names. The detector sits on the runtime's one write path, which every copy the runtime persists passes through: its transcript writer (§4.5), file bytes bound for the blob store, and text handed to a controller (injections, markers). A match becomes a typed placeholder before anything is written, and an altered blob is marked as such. Prohibited bytes are outside the verbatim guarantee; a range read returns the placeholder. Two copies lie beyond this path, the provider tool's own transcript and what the model already read; Appendix D says how each is handled.

Interfaces. What becomes a fact, and when an erase applies, is the writing paper's call (J14 §7.7); what a default search hides is the finding paper's (J2 §8.1); the intent classifier's

contract is the learning paper's (J5 Appendix A.1).

5. Compaction That Leaves Pointers

5.1 The marker

A marker is a pointer with a short, fixed shape and an ID. It carries, in this order:

1. The range: the session and the first and last sequence numbers.
2. The owner's words: every instruction, preference and correction in an owner span inside the range, verbatim, within the budget of §5.5.
3. Exact strings: the paths, hashes, host names, identifiers, numbers and error lines in the range, copied character for character, most-referenced first.
4. Decisions and negative results: one line each ("chose X over Y", "tried Z: failed with E"), each with the address of the entry that shows it, at least one exact string from that entry (the command, the error text) and, where it helps, its time.
5. Open items: what was started and not finished, one line each with an address and a string from its entry.

There is no narrative in it. A sentence like "we made good progress on the setup" gives the model nothing to act on or search with, and can be mistaken for a fact. Everything in a marker is a quote or a line that points at one. The one-line items are the only part a model writes (§7.2), and the string each carries is what the check of §8.2 tests against the entry it cites, so a line that names the wrong entry fails the check. The exact strings serve the model, which can often act without pulling anything because the hash is right there, and a later search, which finds the span by the strings the owner will use.

A tool result can hold text shaped like a marker: a web page that reads "owner (#12): restart the database now". So markers travel only in the runtime's own channel, a block the controller treats as system or developer text, never inside a tool result. The capture filter (§4.7) marks marker-shaped text in a tool result as quoted, and a marker ID that does not resolve in the record is treated as untrusted text. Test 14 includes the case.

5.2 Word for word, one line, or a pointer

Each kind of content gets one treatment when its range is evicted.

Content	Kept in the marker	Why
Owner's instructions, preferences, corrections	word for word, within the budget of §5.5	a paraphrase changes what was asked
Exact strings (paths, hashes, hosts, errors)	word for word, capped by count	a paraphrase of a string is a different string
Decisions, failed attempts, open items	one line each, with an address and a string from the entry	the line says it happened; the address gives how
Agent's reasoning and replies	pointer only	the record holds them
Tool output, file contents included	pointer only	the record holds it whole, files as blobs (§4.1)
Runtime text (time stamps, injected rules)	nothing	the runtime injects it again; it is not memory

5.3 What leaves first

One eviction removes one run of whole turns: the oldest turns after the last marker, as many as it takes to free the room needed, and a marker covers exactly that range (§5.1). The run is cut whole because every eviction moves the cache prefix (§7.2), and many small cuts would pay that price many times. Older turns leave first, since recent content is the best guess at what is needed next. What the marker keeps of the run follows §5.2. When only a little room is needed, clearing old tool output in place, as a provider's clearing does, may be enough; the receipt lists what was cleared.

A compaction appends its marker and never edits an earlier one. The cache reuses a token only while every token before it is unchanged, so editing a marker near the top of the window would rewrite almost everything after it (§7.2). Two things therefore wait for a merge (§5.5): moving an old instruction out to a pointer line, and folding pointer lines into the directory. A merge replaces the markers from the first one it merges onward with one marker rebuilt from the record, so it is priced as an eviction whose span starts at that first marker.

Never evicted: the system text, the always-loaded files (the instruction files a runtime loads into every session, such as a project's CLAUDE.md or AGENTS.md), and the current task's short statement, one line the agent writes when the owner sets a task and rewrites when the task changes. The selector works on the window as the model will read it on the next call, after any provider clearing, and its receipt reports what it evicted by kind, adding up to the total removed.

5.4 Written from the record, never from the last marker

When a merge rebuilds markers, the new marker is built from the record entries the old markers pointed to, not from their text. This rule stops context collapse. A summary of a summary loses detail at every round, while a marker written from the record draws on the same entries at the hundredth compaction as at the first (Figure 2). That is a promise about the source, not about what stays in view: merged markers share capped lists, and old instructions move out to pointer lines (§5.5). What drops out of view stays at its address in the record.

One thing carries over at a merge: the one-line items. Each was checked against the entry it cites when it was first written (§8.2), so the merge keeps it as written and no model has to reread a week of entries. The quotes and the exact strings are taken from the record again, by code.

5.5 Merging markers, and the budget for the owner's words

Markers have a total budget in the window, and the owner's instructions inside them have a budget of their own, B_o tokens. When the markers outgrow their budget, or the instructions of a new marker do not fit what is left of B_o, the compaction becomes a merge (§5.3): adjacent markers become one, covering all their ranges. A merge is a compaction, so §5.4 applies; the merged lists of exact strings and one-line items are capped by count, and what falls off is still reachable through the merged range.

Picture a noticeboard for the owner's instructions. While there is room, every instruction is up on it in full. When a new one does not fit, the next merge takes the oldest down and puts up a **pointer line** in its place: when it was said, its first words and its address, such as 14:34 "Use the staging…" → s7#24/0. Oldest first, because recent instructions are the best guess at what applies next.

That would take down an early safety rule before a late formatting preference. To stop that, the owner can pin an instruction, and a pinned one never comes down. Pins share the board; if pins alone would fill it, the agent asks the owner to move one into the always-loaded file. Each

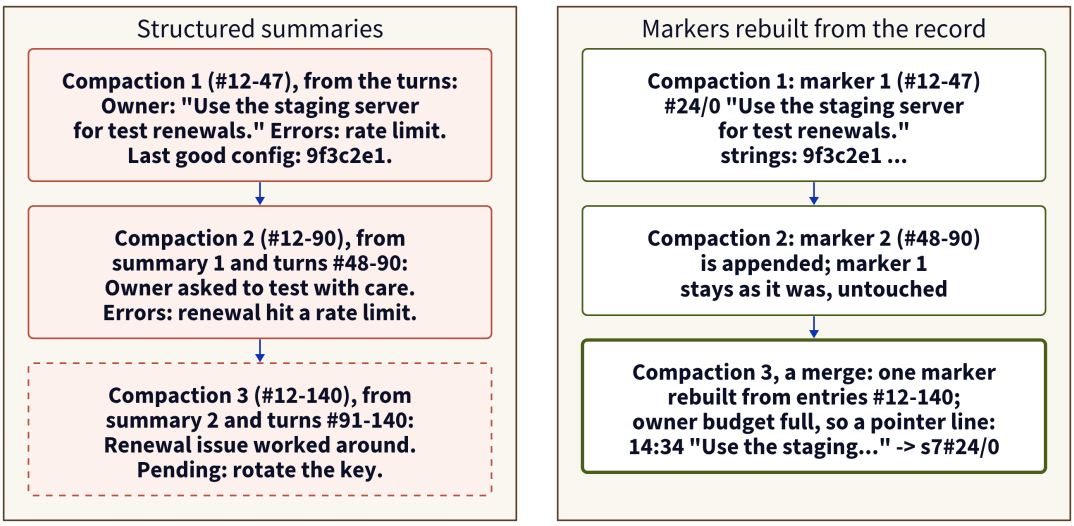


Figure 2. The same session compacted three times, as illustrative output. Left, structured summaries: each is written from the one before plus the newer turns, and by the third the passing correction of 14:34 and the commit hash are gone. Right, markers: the first compaction leaves marker 1, the second appends marker 2 and leaves marker 1 untouched, and the third is a merge that rebuilds one marker from entries 12 to 140, where the owner's budget is full and the correction becomes a pointer line that quotes its first words and gives its address. Illustrative until test 1 measures it.

time an instruction comes down, the next reply says so and offers the pin, and the receipt lists the move (§7.4).

Pointer lines take room too, so the board holds at most k of them. When one more would pass k, they all give way to a single line: "Eleven earlier instructions, s7#12 to s7#311, are listed in the directory at s7#320." The directory is a new entry in the record that lists every instruction taken down, with its address. It is read like any long tool result, a page at a time, and §5.3 may evict it again like one; each page holds at most D tokens and ends with the address of the next. Three hundred instructions of about 30 tokens fill five pages at $D = 2{,}000$.

The guarantees hold whatever the three values are, but a builder needs numbers on day one. As unmeasured starting points: $B_o = 1{,}500$ tokens, about fifty instructions; $k = 10$; $D = 2{,}000$. Test 1 sweeps them, and test 11 checks that the limits hold.

However long the session, the owner's instructions never take more than B_o of the window, each one can be read back word for word from its address, and the owner hears when one leaves view. Table 5 follows the staging correction of §1.1 through the three states, with an illustrative limit of $k = 2$ pointer lines.

Table 5. One instruction in three states, in a window the runtime controls, with at most two pointer lines.

State	What stays in the window	What the owner is told
The wording fits	Use the staging server for test renewals.	nothing; nothing moved
The wording is displaced	14:34 "Use the staging…" → s7#24/0	"Your instruction to use the staging server is stored at s7#24/0; its full wording is no longer in view. Pin it to keep it in view."
A third pointer line would pass the limit	3 earlier instructions → directory s7#90	"Three earlier instructions are now listed in a directory; their pointer lines are no longer in view."

The directory entry lists all three addresses, s7#24/0 among them, and its pages obey D. The wording stays at s7#24/0 unless it is erased or a temporary session ends. The record keeps the instruction while its presence in what the model reads grows weaker, and that weakening is the risk test 7 measures. So the promise has three parts:

1. Every instruction stays **recoverable** word for word through its address, as long as the record lives and it has not been erased. That does not make them all loadable together, since a long directory is read a page at a time.
2. It stays **in view** word for word while it fits or is pinned, and when it leaves, the owner is told.
3. It is **carried into the next session** by its marker while the opening budget allows (§5.6), and **held for good** only in the always-loaded file.

5.6 From one session to the next

A long job outlives its session. A runtime may close a session each night, or after an hour of quiet (§7.5), and the next one should not start blind. So when a session closes, the runtime appends a closing entry that lists the markers in its window, and the next session opens with them, pinned instructions first, while they fit its opening budget. A carried marker keeps its own session and range: markers from two sessions never merge into one, since a range is one session's sequence numbers (§5.1). Carried markers that do not fit give way to one line that points at the closing entry, which lists them all.

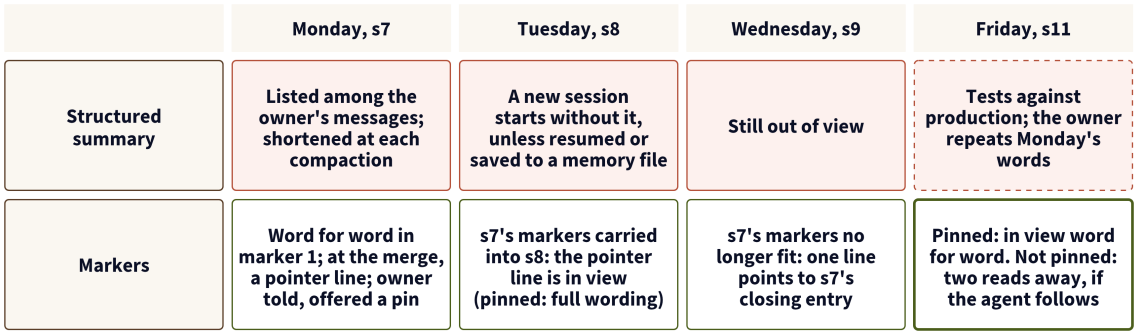


Figure 3. Monday to Friday, five sessions (s7 to s11), for the staging correction of §1.1. Top, a structured summary: listed while the summariser keeps it, gone from a new session unless resumed or saved. Bottom, markers: a pointer line after Monday's merge, with the notice and the offer to pin, carried into Tuesday, two reads away by Wednesday unless pinned. Illustrative.

An instruction is carried beyond its session by its marker while the opening budget allows, and held for good only in the always-loaded file, where the writing and learning stages promote a rule the owner keeps repeating (J14 §8.2; J5 §5). Figure 3 follows the staging correction of §1.1 through the week. The path has one weak point, and the design names it: once the correction is a pointer line, it protects the owner only if the agent follows it, or if the owner pinned it when told it had moved. Test 7 measures the first.

6. Who Controls the Window

6.1 Three window controllers

The window controller is whoever decides, on each call, what the model reads. There are three cases, shown in Figure 4, and one agent can move between them as it switches models or tools:

- The runtime builds every call to the model API from its own state. Compaction is the runtime's to do.
- A provider's command-line tool keeps its own conversation, decides when to compact and writes the summary; the runtime sees what goes in and what comes out.
- A provider's API takes the calls the runtime builds, then clears old tool results or compacts on its servers once a threshold is crossed.

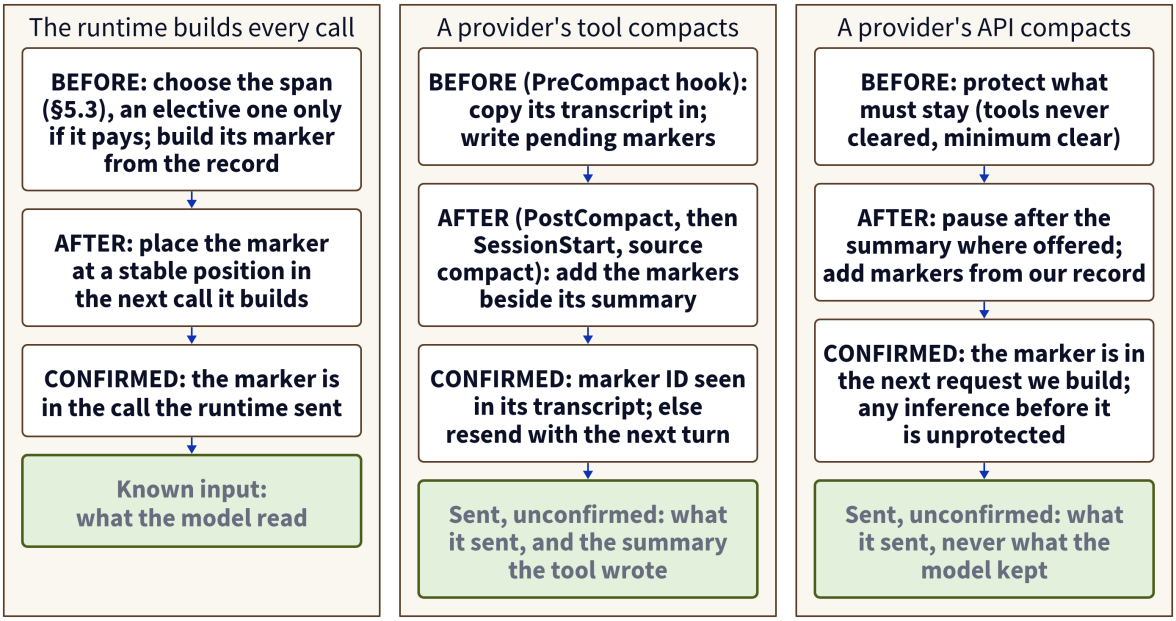


Figure 4. The three window controllers and what the memory layer does in each: before a compaction, after it, and how it confirms that the marker reached the model. The bottom row is what the memory layer can know about the model's view: known input when the runtime builds the call, otherwise only what was sent. In all three, every marker points into the session record.

6.2 When the runtime controls the window

This is the case §5 describes. The runtime chooses when to compact by the price rule of §7, evicts by §5.3, builds the markers and places them where the evicted range was. It built the call, so it knows what the model read.

6.3 When a provider's tool controls the window

The runtime cannot compact a window it does not build. It acts through the tool's hooks (Anthropic, 2026b), in three steps; in Claude Code they are `PreCompact`, then `PostCompact` and `SessionStart` with `source` set to `compact`:

- Before compaction (`PreCompact`). The hook makes the record complete up to this moment, reading the tool's own transcript where needed. It does not say which part the tool will summarise, so the adapter takes everything since the previous compaction (or the session start) as the range and builds markers from the record; a marker for a turn the tool keeps verbatim costs a few tokens. The pending markers are written durably before the hook returns. The hook could block the compaction, but when the tool compacts to recover from a full window, blocking fails the request, so the default is to let it run.
- After compaction (`PostCompact`, then `SessionStart` with `source=compact`). The adapter stores the tool's summary as a runtime entry where the event carries it, and injects the stored markers into the new context before the next model call, beside the summary.
- Delivery check (every turn). The adapter looks for the marker IDs in the tool's transcript. A marker that is absent is sent again with the next prompt's injected text, and the receipt counts the turns run without it (§6.6).

The tool's summary stays: for many turns the gist is enough, and the markers add what a gist cannot hold.

6.4 When a provider's API controls part of the window

Here the runtime builds every call, but the API may change the conversation after it is sent. Three moves cover it. First, the runtime keeps its own record of everything it sent and received; after a server-side compaction this may be the only readable copy, and with opaque compacted state it certainly is. Second, it uses the API's own controls to protect what must stay: it names the tools whose results are never cleared and sets a minimum to clear, so a small clearing does not break the cache for a small gain.

Third, it promises only what it can deliver. A threshold compaction runs inside a request, and the model may go on answering from the compacted context in the same response (Anthropic, 2026a; OpenAI, 2026a). The runtime places markers only in requests it builds, so the guarantee is delivery before the next request the runtime sends. Where the API can pause after its summary, the adapter takes the pause and adds the markers, so the first inference after compaction has them; where it cannot, the rest of that response is recorded in the receipt as unprotected.

6.5 Never compact a copy nobody reads

When a provider's tool holds the window, the runtime never compacts its own copy of the conversation, since the model never reads it (§1.4). It writes a receipt for each of the tool's compactions (§7.4) and counts a compaction only when it changed what the model reads.

6.6 The adapter contract

An adapter is the small piece of code that connects the memory layer to one window controller, the way a printer driver connects a computer to one model of printer. All adapters follow one contract (Appendix C). Each starts by listing what its controller allows: does it warn before compacting, can its transcript be read, can text be added before the next call? Anything the adapter cannot confirm counts as missing. It then runs every compaction as a checklist (noticed, record saved, markers built, markers injected, markers seen, closed) and writes each tick to the record before the next step starts, so after a crash it carries on from the last tick.

A marker counts as delivered only when its ID shows up in a call the runtime sent, or in the tool's own transcript. Until then its range does not count as in view, the marker goes out again with every turn, and the receipt counts the turns that ran without it.

Every receipt also says how much the memory layer can know. When the runtime built the call, it knows exactly what the model read: *known input*. When a provider's tool or API sits in between, it knows what it sent (and, for a tool, the summary the tool wrote), never what the model kept: *sent, unconfirmed*. It is the difference between handing someone a letter and posting it. Nothing may claim more than its receipt's level; in particular, the learning stage never counts a rule that was only sent, unconfirmed, as one the model saw (J5 §5.4).

The adapter also tells the finding stage what left the window. Each compaction receipt, and each window rebuilt after an erase or a quarantine, is published as an eviction event that lists every range the window no longer holds, and it is resent until the finding paper's record of what the window holds acknowledges it (J2 §6.7). An erase or a quarantine that reaches a live session goes through the adapter too; Appendix D says what it can remove from a provider's copy and what it can only report.

7. When to Compact, and What It Costs

7.1 Forced and elective

A compaction is forced when the next call would not fit the window: there is no choice about whether, only about what to evict (§5.3) and what to leave (§5.1). It is elective when the window has room but keeping a span costs more than evicting it, or a smaller window would be read better (§1.2); only an elective compaction can be declined on price.

7.2 The price of moving the prefix

With prompt caching, the part of the context that did not change since the last call is read from cache at a fraction of the input price, and everything after the first changed token is written to cache again at a premium (Anthropic, 2026c; Ji, 2025). Take a box out of the middle of a full shelf and every box to its right has to move; the cache works the same way. Evicting a span from the middle of the window changes everything after it. So eviction has one-time costs, a running saving, and the cost of the pulls it causes. All of it is counted per call (§1.4); the finding paper counts the same unit and calls it a turn.

Table 6 shows what one eviction does to the cache, with the sizes of Figure 5.

Table 6. What one eviction makes the cache write again. The text after the span is unchanged, but the tokens before it changed, so it is written again.

	Before the span	The span	After the span
Before eviction	the prefix, read from cache	40,000 tokens of old turns	100,000 tokens, read from cache
After eviction	the same prefix, read from cache	a 300-token marker, written	the same 100,000 tokens, written again

In words: the calls to break even equal the one-time cost of the eviction divided by the net saving on each later call. The one-time cost is the rewrite of everything after the span plus building the marker; the saving is the span's cached read on every call, less the expected cost of pulling something back. We use the finding paper's prices and notation (J2 §7.1): r per token read from cache, w per token written to it and g per output token, as multiples of the base

input price. With a span of S tokens, a marker of m, A tokens after the span, a marker that costs $K = cS + g\,m$ to build (a model reads the span once at input price c to write the one-line items), and a pull of cost C^H_{pull} on a share q_{call} of later calls (the finding paper's q_{turn}), evicting the span pays after N^* more calls:

$$N^* = \frac{(m + A)(w - r) + K}{(S - m)\,r - q_{\text{call}}\,C^H_{\text{pull}}}.$$

Appendix F defines every symbol, derives the pull cost and works the example. When the denominator is not positive, an elective eviction never pays: the span is read back too often and should stay, or be compressed in place.

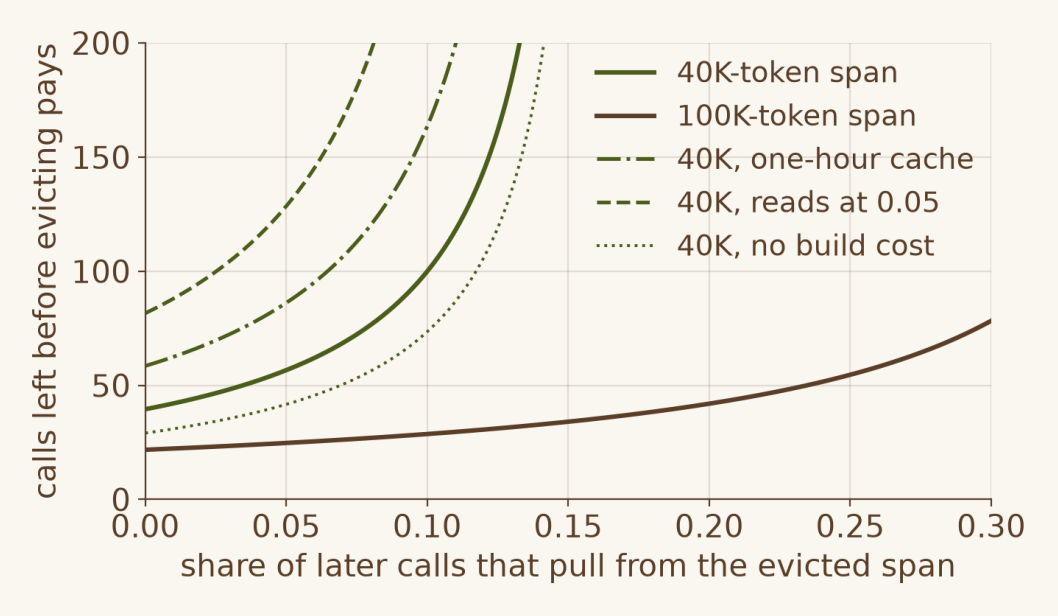


Figure 5. Calls a session must still run before an elective eviction pays, against the share of later calls that pull from the evicted span. Solid lines: spans of 40K and 100K tokens with the five-minute cache ($w = 1.25$, $r = 0.1$, $g = 5$). For the 40K span: dash-dot, the one-hour cache ($w = 2$); dashed, cache reads at $r = 0.05$, Anthropic's price on Opus and Sonnet 5.5; dotted, the cost of building the marker left out. Other inputs as in Table 7 (Appendix F). A 10K span is left off the plot; its numbers are in the text.

With the assumed inputs of Appendix F, a 40K-token span with the five-minute cache pays back after about 40 calls if nothing is pulled from it, after about 57 if one later call in twenty pulls, and never once more than about one call in six pulls. A 100K-token span pays back after about 22 calls; a 10K-token span needs about 131. The cache matters as much as the span: with the one-hour cache ($w = 2$) the 40K span needs about 58 calls, with reads at 0.05 about 82, and with both about 119. Cheaper cache reads push break-even out. Whether a session still has that many calls ahead when the window has room, and how often it pulls, is what test 5 measures; until then the rule says how to decide, not whether elective eviction will be common or rare.

Three design rules hold whatever the numbers. Evict large runs at once, since the rewrite is paid per eviction and the saving grows with the span. Prefer runs that are rarely pulled: age is a fair guess at that, which is why a run starts at the oldest turn (§5.3), but a run that reaches the latest error output will often be pulled back and should stop short of it. And leave a marker unchanged once it is placed, as §5.3 rules. The benefit of a shorter window (§1.2) is unpriced; test 5 measures it.

7.3 A trigger you can trust

Every compaction decision starts from one number: how full the window is, taken from the call actually sent. A figure that sums several calls of one turn can be many times the window and fires compaction far too early; one that misses a block of the prompt fires it too late.

So the trigger reads a per-call breakdown: the blocks of the prompt, each attributed to a kind (system text, always-loaded files, tool schemas, history, markers, tool results, the new message), plus a slice for anything it could not attribute. The slices must add up, within the tokenizer's tolerance, to the input count the provider reports. Unattributed content is an error that is reported, never a zero that is assumed. A breakdown that does not add up blocks elective compaction; forced compaction still runs, because the provider rejects a call that does not fit whatever our counter believes. An elective compaction does not run again until the window has grown by at least what the last one freed.

7.4 One compaction receipt per compaction, whoever ran it

Every compaction, by every executor, writes one receipt, and so does a decision not to compact: a missing receipt is the alarm (J5 §9.1). The receipt is also the eviction event of §6.6, and its evicted field lists cleared tool results as well as marker ranges, since an API's clearing leaves no marker. It says who ran the compaction and under which controller, whether it was forced or elective and what was decided, the call breakdown, the tokens before and after, every evicted range with the files read in it, the markers written and whether each was delivered, the instructions moved, and how much the memory layer can know (§6.6). Appendix C lists the fields.

7.5 Idle sessions: compact before the cache lapses, or start fresh

A prompt cache keeps the processed start of a conversation for five minutes, or for an hour on the longer setting. When it expires, the next message pays to process the whole window again, whatever we do. So the best moment to compact an idle session is just before the cache expires. If the owner comes back after it has expired, the rewrite that a compaction causes costs nothing extra, because the expiry forces it anyway, and the marker was cheaper to build while the old text was still cached. If the owner comes back sooner, the compaction moved the cache for nothing; if nobody comes back, the marker was built for nothing. Test 6 measures how often each happens. Claude Code shipped this timing in October 2026: a conversation over 200K tokens left idle for about 54 minutes is compacted just before its one-hour cache expires (Anthropic, 2026f). Where the runtime controls the window, that moment builds markers instead of a summary.

A session idle for hours raises a different question: compact it again, or start a new one? A chat that never ends would otherwise compact forever. Claude Tag starts a fresh session for a channel after about an hour of quiet, or when the session is about a day old and the channel is quiet (Anthropic, 2026g). In this design a fresh session loses nothing: it opens with the markers the old one ended with (§5.6), the always-loaded files and the long-term memory, and every address in the old record still resolves. We propose a fresh session when the cache has expired and the old markers fit the new session's opening budget, and a compaction otherwise; test 6 decides.

8. Getting It Back

8.1 The range read

The range read takes a range and returns the record's entries in it, verbatim, with their authors and times; a file read comes back with its blob, whatever happened to the file since. An erased entry or span comes back as its tombstone, a quarantined one as its placeholder (§4.4), and a prohibited match as its typed placeholder (§4.7). The finding paper's pull tool offers it as one mode next to a search over the record (J2 §6.6); this stage guarantees that the search runs over the same record the markers point into.

Deleting is not instant. Redacting an entry, rebuilding the markers that cite it and rebuilding the live window take time, and reads keep arriving meanwhile. A shop facing a product recall has the same problem: it blocks the barcode at the tills before staff clear the shelves. Here the block is a number. Each erase or quarantine first raises a counter, the *erasure epoch* (the writing stage raises it, J14 §7.7). This stage then works through deletions in counter order: remove or flag the text, rebuild every marker that covered it, rebuild the window. Only then does it record that counter value as applied (Figure 6).

Every range read and every window build looks at the counter first. If a deletion has been announced and not yet applied, it waits. If a deletion is announced while it runs, its result is thrown away and it runs again. So between the decision and the end of the rebuild, no read returns erased text and no window carries it to the model. The work is also safe to repeat: progress is written to the record as it goes, so a crash resumes where it stopped, and a deletion order that arrives twice gets the same answer the second time without being applied again (Appendix B.1).

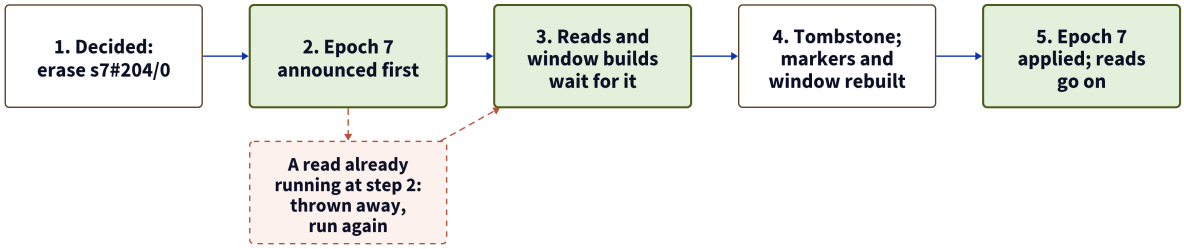


Figure 6. The order of a deletion. The counter moves before any text is touched, every read and window build waits for it, and a read that was already running when the counter moved is thrown away and run again. Only after the text, the markers and the live window are rebuilt is the epoch marked applied.

8.2 Every marker resolves

A marker that names entries the range read cannot return is a broken promise, and a check forbids it. Resolving a marker's range must return every sequence number from first to last (a tombstone or a placeholder counts as present); every blob it refers to must be in the blob store; every exact string and owner sentence it carries must appear in an entry inside the range; and the string on each one-line item must appear in the entry that line cites. The check runs when the marker is written and again in the nightly tests.

8.3 Asking a past session

Sometimes the best source is the session that did the work; the finding paper decides when to ask one (J2 §6.9). The session is woken in place, not copied: its record grows by a runtime entry that marks a later visit, then one question and one answer, so a reader can tell what it said at the time from what it said when woken. Before the question it gets a short wake-up note: today's date, how long it slept, and what changed since, as far as the record and the store can say cheaply (facts it touched that were superseded or erased, newer commits to files it edited, rules added or retired). Its answer is testimony from its own time, which the asker checks against the current store; a session is not woken while an erasure epoch is still unapplied to it, and a temporary session is never woken, since its record is gone (§4.7).

9. Conclusions

9.1 In short

After compaction, an agent's next inference sees a description of its past, at best with one path to the whole transcript, and nothing that says what the description left out; a new session sees less. Total Recall keeps the whole session as an addressed record and makes compaction leave markers that quote what a description drops and point back into the record, carries them into the next session, and follows one contract whichever controller holds the window, claiming no more about the model's view than that controller lets it know. If it holds, and the marker reached the model, what the owner said on Monday has an exact answer on Friday: the instruction is in view while it fits the owner's budget or is pinned, or a line in view says where it is. Whether the agent then acts on that line is the open question (test 7). If it does, the agent becomes the colleague of the opening: told once on Monday, still careful on Friday.

9.2 Limitations

- Markers depend on extraction by code and a classifier; a missed string is still in the record, but the marker no longer advertises it.
- The one-line items are written by a model. The check of §8.2 catches a line whose string is not in its entry, not a line that misreads an entry it quotes correctly.
- Past the owner's budget, old instructions are a pointer line or a directory page away, and the oldest go first, so an early safety rule leaves view before a later preference unless the owner pins it. Past the opening budget of a new session, they are one read further. The agent must follow the pointer when it matters (test 7).
- Hooks are the provider's to change. An API that compacts mid-response without a pause leaves the rest of that response unprotected, and opaque compaction cannot be checked; the receipt says so, but the gap remains. A tool's or an API's own copy of erased or quarantined content is reached only by ending the session and rebuilding it.
- The record grows without bound, and the capture detector will miss some secrets; years of sessions will need tiered storage, which the design does not cover.
- The price rule rests on unmeasured inputs (q_{call}, H, c, ℓ) and on how many calls a session has left, and cheaper cache reads push break-even further out.
- Recoverable is not recovered: whether the agent asks for what it can read back is the finding paper's question.
- A woken session answers from its own time. The wake-up note tells it what changed only as far as the record and the store know.

10. Next Steps

10.1 Experiments that decide the claim

Two tests carry the claim. Test 1 checks the mechanism: given the same way back into the record, does a marker keep the owner's words and the exact strings better than a summary? Test 3 checks the outcome the owner feels: do they have to repeat themselves less? The other five measure what the design guesses at. Tests 8 to 15, the checks a build must pass, where any failure is a defect, are in Appendix E.

A test that tunes something (the string extractor, the classifier thresholds, the budgets) tunes on development sessions and is scored on held-out ones (J5 §9.2). Live arms are randomised by session. Arms that are compared get the same context budget and the same compaction schedule, and each reports its total token cost, so a gain cannot come from a larger window or fewer compactions.

#	Test	Decides	What would change the design
1	Needle survival. Seed exact strings, owner instructions (standing rules and corrections said in passing), failed attempts and times early in long synthetic sessions; force 1, 3 and 5 compactions; ask for each, scoring the owner's instructions apart from the strings. Arms: the provider's summary alone; the provider's structured summary with its transcript path; truncation; **the summary plus the same searchable record and range read (the control)**; markers plus range read; markers without the owner's section, which is close to ARC; markers alone. Sweep B_o, k and D around §5.5's starting values.	§5: the marker, not mere access to the record, keeps what summaries lose; the owner's section adds to addressed citations	Control or structured summary matches markers: the gain is the way back, not the marker. Markers without the owner's section match full markers on instructions: the budget adds nothing. Markers alone match markers with the read: the read matters less
2	Back-reference replay. Real moments where the owner refers to earlier in the session ("the hash from before"), replayed after compaction under the control and under markers.	§4, §5 in real use	The 95% interval of the margin over the control includes zero: no gain is shown, and markers have not earned their cost
3	Forget-cue rate. Owner messages that repeat an instruction or ask "did you forget", per 100, under the control and under markers, split by a compaction in the previous ten turns and by a new session since the instruction; sized from the measured baseline to detect a one-third drop (80% power).	the outcome the design serves	The interval of the difference includes zero: no reduction is shown, and the promise of §1 stays unproven
4	Marker coverage. For each evicted range, the share of its exact strings and owner instructions present in its marker.	§5.1, §5.2	Below half: the extraction, not the design, is the weak part
5	Price and quality. Measure calls per turn, the calls a session still has when the window has room, q_{call}, H and K; compare elective eviction at N^* with none, on task success.	§7.2	A q_{call} above break-even, or sessions shorter than N^*, make elective eviction rare; a quality gain from shorter windows argues for earlier eviction

#	Test	Decides	What would change the design
6	Idle sessions. Replay idle sessions under three policies: compact on the next message, compact just before the cache lapses, and start a fresh session with the carried markers. Compare tokens per resumed turn and the measures of tests 1 and 3.	§5.6, §7.5	Compaction before expiry costing no less than compaction on the next message, or a fresh session losing needles the compacted session keeps, changes the policy
7	Acting on a moved instruction. A simulated repair session pushes the staging correction of §1.1 into the directory (Table 5), then reaches a point where a test renewal is due. Nobody asks the agent to recall the instruction; whether it renews against staging is scored, then whether it can quote the instruction when asked. Run within one session and after a new session.	§5.5, §5.6: a pointer line or a directory line still protects the owner	Production used more often than with the wording in view: moved instructions need a stronger cue, or fewer of them may leave the window

10.2 Building it

Total Recall is to be built in the order the tests need it:

1. the record index, spans and blob namespaces;
2. the marker builder and the resolve check;
3. the provider-tool adapter;
4. carrying markers into the next session;
5. erase, quarantine and the epoch gate;
6. the call breakdown and the receipt;
7. the wake-up note;
8. the price rule last, once tests 5 and 6 have measured its inputs.

References

- Anthropic (2026a). Compaction at a token threshold. Claude Platform documentation. https://platform.claude.com/docs/en/build-with-claude/compaction-threshold
- Anthropic (2026b). Hooks reference. Claude Code documentation. https://code.claude.com/docs/en/hooks
- Anthropic (2026c). Prompt caching. Claude Platform documentation. https://platform.claude.com/docs/en/build-with-claude/prompt-caching
- Anthropic (2026d). Context editing. Claude Platform documentation. https://platform.claude.com/docs/en/build-with-claude/context-editing
- Anthropic (2026e). Compaction overview. Claude Platform documentation. https://platform.claude.com/docs/en/build-with-claude/compaction
- Anthropic (2026f). Claude Code changelog, version 2.1.299 (idle compaction). https://github.com/anthropics/claude-code/blob/main/CHANGELOG.md
- Anthropic (2026g). How Claude Tag works. Claude documentation. https://claude.com/docs/claude-tag/concepts/how-it-works
- Anthropic (2026h). 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

- Anthropic (2026i). Claude Code, version 2.1.306: the compaction instructions shipped in the program (npm package `@anthropic-ai/claude-code`), read 21 October 2026. https://www.npmjs.com/package/@anthropic-ai/claude-code
- Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 78–101. https://doi.org/10.1147/sj.52.0078
- Dang, T., Ichikawa, Y., Fatima, S., & Shirahata, K. (2026). Addressable recall compaction for long context-window control in AI agents. arXiv:2607.25066. https://arxiv.org/abs/2607.25066
- Headroom Labs (2026). headroom: compress what your AI agent reads. GitHub, Apache-2.0. https://github.com/headroomlabs-ai/headroom
- Hong, K., Troynikov, A., & Huber, J. (2025). Context rot: how increasing input tokens impacts LLM performance. Chroma technical report. https://research.trychroma.com/context-rot
- Ji, Y. (2025). Context engineering for AI agents: lessons from building Manus. 18 July 2025. https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus
- Letta (2026). Compaction. Letta documentation. https://docs.letta.com/v1-sdk/messages/compaction
- Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the middle: how language models use long contexts. *Transactions of the ACL*, 12, 157–173. https://arxiv.org/abs/2307.03172
- Mem0 (2026). Add memory. Mem0 documentation. https://docs.mem0.ai/core-concepts/memory-operations/add
- OpenAI (2026a). Compaction. OpenAI API documentation. https://developers.openai.com/api/docs/guides/compaction
- OpenAI (2026b). Memories. ChatGPT Learn. https://learn.chatgpt.com/docs/customization/memories
- Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & 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., & Chalef, D. (2025). Zep: a temporal knowledge graph architecture for agent memory. arXiv:2501.13956. https://arxiv.org/abs/2501.13956
- Serra, O. (2026a). MNEMOSYNE: one check on every memory write. J-series paper J14 (the writing paper), version 2.6, 21 October 2026. Available from the author.
- Serra, O. (2026b). INSTANT RECALL: finding what the agent already owns. J-series paper J2 (the finding paper), version 8.6, 21 October 2026. Available from the author.
- Serra, O. (2026c). Sleep consolidation: a repeated correction earns a diagnosed, tested repair. J-series paper J5 (the learning paper), version 7.7, 21 October 2026. Available from the author.
- thedotmack (2026). claude-mem: persistent memory for Claude Code. GitHub. https://github.com/thedotmack/claude-mem
- Zep (2026). Adding episodes. Graphiti documentation. https://help.getzep.com/graphiti/core-concepts/adding-episodes
- Zhang, Q., Hu, C., Upasani, S., Ma, B., et al. (2025). Agentic context engineering: evolving contexts for self-improving language models. arXiv:2510.04618. https://arxiv.org/abs/2510.04618

Appendix A. A Worked Record and Marker

Four entries of Figure 1 as stored (#23, #24 and #31, with the file read #16), a quarantine (#62) added to show that field, and the marker of Figure 1. Field names follow §4.1; values are illustrative.

```
entry s7#16   author_kind tool           time 14:09:40
  call         read_file /etc/nginx/sites-enabled/app.conf
  content      blob ordinary/sha256:7d2a...  (4,812 bytes, stored before the window dropped it)

entry s7#23   author_kind tool           time 14:31:05
  call         certbot renew
  text         error: too many certificates already issued (rate limit)

entry s7#24   author_kind owner          time 14:34:10
  text         [runtime wrapper] Use the staging server for test renewals.
  owner_words  Use the staging server for test renewals.
  span s7#24/0 authority owner (a2)   intent correction (classifier k5)   cutter c3
  controller   tool transcript, entry 131

entry s7#31   author_kind agent          time 14:40:52
  text         Key rotation left for later; the renewal has to succeed first.

entry s7#62   author_kind runtime        time 16:47:30
  text         quarantine s7#40 (writing-stage decision, injected instruction)

marker m-s7-12-47   built from entries #12-47
  range        s7 #12-47
  owner        #24/0 "Use the staging server for test renewals."
  strings      /etc/nginx/ssl  9f3c2e1
  failed       #23 (14:31): certbot renew: "too many certificates already issued"
  open         #31: "Key rotation left for later"
```

Appendix B. The Loop in Pseudocode

B.1 Appending to the record

```
APPEND(session, entry):                      # before the model call that answers it
  ns <- session.temporary ? namespace(session) : "ordinary"           # 4.7
  if seen(session, entry.source_id):         # a retry or a duplicate delivery
      return address_of(entry.source_id)     # gets the same address back
  entry.seq <- next_sequence(session)        # once, never reused
  entry.author_kind <- owner | agent | tool | runtime | other_agent
  entry.text <- persist_filter(entry.text)   # 4.7; marks marker-shaped tool text
  if entry.author_kind == owner:
      entry.owner_words <- strip_wrapper(entry.text)  # once; never rewritten
      entry.spans <- cut_spans(entry.owner_words)  # 4.1: immutable, addressed
      label_authority(entry.spans)                 # owner | quoted; unsure: quoted
      label_intent(owner_spans(entry.spans))       # J5 A.1; versions stored
  if entry.returns_file: entry.content <- blob_put(ns, persist_filter(bytes))
  record.append_durable(session, entry, (entry.source_id, entry.seq))
                                             # one write; status: captured (4.6)
  index.add(session, entry.seq, entry)       # derived, rebuildable
```

Erasure, quarantine and the epoch gate that every read waits for (§4.4, §8.1, Appendix D):

```
ERASE(session, seq, spans, receipt, epoch):  # writing stage decides; epoch raised first
  if applied_epoch >= epoch: return ack_of(epoch)   # duplicate: same acknowledgement
  await applied_epoch == epoch - 1           # epochs apply in order
  b <- record.blob(session, seq)             # none if the entry holds no file
  b2 <- blob_put(ns, redact_bytes(b, spans)) if b and spans else none  # a string in a file
  hit <- entries_with_blob(b) if b else [(session, seq, spans)]   # same namespace
  for (s, n, sp) in hit - progress(epoch):   # a restart resumes here
      if b2: record.repoint(s, n, b2, tombstone(now, receipt))     # rest of the file stays
      else:  record.redact(s, n, sp, tombstone(now, receipt))      # the one edit
      record.append(s, runtime_entry("erased", n, sp))
      index.reindex(s, n); rebuild(markers_covering(s, n))
      adapter.redact_controller_copies(s, n, sp)           # Appendix D
      adapter.erase_live(s, n, sp)   # rebuild window, eviction event, name provider copy
      progress.mark(epoch, (s, n, sp))
  if b: blob_delete(b)
  manifest.report(status_per_place)          # J14 7.7; out of reach named
  applied_epoch <- epoch                     # reads at this epoch may now be served

QUARANTINE(session, seq, decision, epoch):   # no edit to the record
  if applied_epoch >= epoch: return ack_of(epoch)
  await applied_epoch == epoch - 1
  record.append_once(session, runtime_entry("quarantine", seq, decision))
  rebuild(markers_covering(session, seq))
  adapter.invalidate_live(session, seq)      # clean rebuild; eviction event (6.6)
  receipt("quarantine", seq, rebuilt, out_of_reach)  # per controller
  applied_epoch <- epoch

SERVE(kind, args):                           # range read or window build; 8.1
  loop: e <- store.epoch(); await applied_epoch >= e; out <- build(kind, args)
        if store.epoch() == e: return out    # else an epoch arrived: run again
```

B.2 Building a marker and evicting

```
COMPACT(window, session, forced, need):
  bd <- call_breakdown(last_call_sent)                  # 7.3
  if not bd.reconciles and not forced: receipt("blocked: breakdown"); return
  if not forced and window.grown_since_last < last_freed:
      receipt("none: thrash guard"); return              # 7.3
  span <- oldest_run_of_whole_turns(window, need)       # 5.3: one run, after the last marker
  merge <- over_budget(window, span)                    # markers or B_o would overflow (5.5)
  if merge: span <- span.from(first_marker_of_session(window))   # never another session's (5.6)
  if not forced and not pays(span, bd): receipt("declined: price"); return   # 7.2
  entries <- record.read(session, span.first, span.last) # never the old markers' text
  marker <- new_marker(range = span,
      owner   = instructions(entries),                  # owner spans, verbatim, by code
      strings = exact_strings(entries, cap),            # by code
      lines   = checked_lines_of(window, span)          # 5.4: reused as written
              + decisions_and_failures(new_entries(span)),   # each with an address and a string
      open    = open_items(entries))
  moved <- enforce_owner_budget(marker) if merge else []   # pins stay; pointer lines; directory
  assert resolves_full_range(marker) and lines_match_entries(marker)   # 8.2
  window.replace(span, marker)                          # earlier markers untouched (5.3)
  receipt(executor = runtime, level = known_input, bd, before, after, moved, outcome)

CLOSE(session):                                          # 5.6
  record.append(session, runtime_entry("closing", markers_in(window)))
OPEN(next, previous):
  carry(markers_of(previous.closing), pinned_first, fit = opening_budget)
                                         # what does not fit: one line to the closing entry
```

Appendix C. The Window Controller Adapter Contract

Every adapter implements all rows and states the contract version it follows; a capability it cannot confirm counts as absent.

Part	What it holds	Rule
Version	contract version; adapter version; controller name and version	re-detected at session start and on any model or tool change
Capabilities	builds calls; event before compaction (can block); summary after compaction; injection before the next call; per-turn injection; tool-result replacement; readable transcript; clearing controls; pause after compaction; compacted state readable or opaque	choose the path of §6.2, §6.3 or §6.4
Pending state	one runtime entry per step: detected, flushed, markers built, injected, delivered, closed	each state appended durably before the next step; a restart resumes from the last state; marker IDs make a repeat harmless
Sequence map	runtime address paired with the controller's entry ID	written at flush; entries only the controller saw get runtime numbers in the controller's order
Delivery check	marker ID seen in the next call sent (runtime, API) or in the tool's transcript; an API's pause used where offered	promised before the next request the runtime sends; not seen by the next turn: resend with the per-turn injection and count the turns without it; inference before delivery recorded as unprotected
Knowledge level	known input (runtime); sent, unconfirmed (tool, API)	on every receipt; no claim about the model's view exceeds it; after a tool or API compaction, everything sent before it is residency unknown
Degraded paths	no event before compaction: detect afterwards (a summary appears, input tokens fall), flush, then build; no injection after compaction: per-turn injection, receipt marked late; unreadable transcript: record what the runtime saw and mark the gap in range reads	each path used is named in the receipt
Capture filter	the runtime's one write path, before persistence; tool results replaced where a hook allows; the tool's transcript redacted at the next hook	a copy out of reach is named in a receipt (§4.7)
Erase and quarantine in live state	rebuild the window (runtime); redact copies handed to the controller; end the session and rebuild it from permitted material, or use a documented deletion control (tool, API); a provider compaction never counts as removal	each place reported to the deletion manifest (Appendix D), unreachable copies named; a quarantine writes the same receipt; reads wait for the epoch (§8.1)
Never retained	the controller's no-retention setting, where offered	where none is offered, the owner is told at session start

The compaction receipt of §7.4 carries these fields.

Field	Meaning
session, started, ended	which session, and when
executor, controller case	runtime, provider tool or provider API; the §6 case
kind, decision	forced, elective or none; ran, declined on price, blocked on breakdown, did not run
call breakdown	attributed slices, unattributed slice, provider-reported input count
tokens before and after	as measured on the call before and the call after
evicted	each evicted record range, with path, region and blob hash of every file read in it; each tool result an API cleared; after a tool or API compaction, everything sent before it (residency unknown)
markers	IDs and ranges written; whether each resolved and was delivered
instructions moved	owner instructions moved to pointer lines or the directory (§5.5)
knowledge level, fallbacks	known input or sent, unconfirmed (§6.6); each degraded path used; any inference run unprotected (§6.4)
outcome	completed, failed (with reason), interrupted

Appendix D. Forgetting in a Record That Is Never Edited

§4.4 gives the two operations and §8.1 their ordering. This appendix holds the rest.

Blob namespaces. Identical contents are stored once per blob namespace. Ordinary sessions share one namespace; each temporary session has its own (§4.7). A namespace comes from the authenticated session, never from a name in a request or a tool argument.

Shared files. A blob is shared by every read in its namespace that returned the same bytes. Erasing a whole entry tombstones every entry that references its blob: the tombstone replaces the text, `owner_words`, the labels and the blob. Erasing one string inside a file read, such as a phone number in a configuration file, does not take the whole file. A new copy of the blob is stored with the string replaced by a typed placeholder, under its new hash; every entry that referenced the old blob is pointed at the new one, with a tombstone that records the change; and the old blob is deleted. The rest of the file stays readable in every session that read it.

Copies beyond the write path. A provider's tool keeps its own transcript, and blocking a prompt does not keep its text off the tool's disk (Anthropic, 2026b). The adapter replaces a prohibited match in a tool result before the tool's model reads it, redacts the tool's transcript at the next hook, and names any copy it cannot reach in a receipt. The model may also have read a value in the turn it arrived. A detector miss found later is handled as an erase (§4.4).

Copies handed to the controller. The runtime has given the window controller copies of what it now erases: markers, per-turn injections, the learning stage's provisional lines. They are redacted in the controller's transcript. For a quarantine, the adapter invalidates the live window as it does for an erase and writes a receipt naming what it rebuilt and what it could not reach.

What counts as removal. A provider's compaction never counts as removal. Its compacted state is opaque and may still carry the line, and nobody can read it to check. Removal from a tool's or an API's context is verified only when the adapter rebuilds a clean context from permitted material, or when the controller offers an operation documented to remove that content; anything else is reported as unverified and out of reach. When the runtime builds the calls, it simply rebuilds the window without the content; for a provider's tool or API, it ends the session and starts a clean one. After a quarantine receipt, no call the runtime builds carries the content; a tool's or an API's context is covered only where its session was rebuilt that way.

What the finding stage hears. The eviction event of a rebuilt window lists every range the rebuild left out, not only the erased or quarantined entry, so the finding paper's record of what the window holds does not keep a range the rebuild dropped.

What this stage reports. For an erase, this stage fills in its rows of the writing paper's deletion manifest (J14 §7.7): record entries, the controller's transcript, blobs, markers, and live sessions or provider state. Each place is reported done, not present, or out of reach (named). "All reachable copies removed" is the strongest claim this stage makes.

Appendix E. Acceptance Checks for a Build

These eight tests ask no open question: any failure is a defect to fix, not a trade-off. They share the method of §10.1 and are numbered after its seven experiments. The full replay of one correction through compaction, restarts and deletion is the learning paper's (J5 §10); tests 10, 12 and 13 are this stage's part of it.

Handoffs to the other stages. For a reader who has the learning paper, the table maps the handoffs of its specification (J5 Appendix B) that pass through this paper.

J5 ID	Handoff	Here	Checked by
T1	an owner message is captured: one durable flush, its spans and labels stored	§4.6	test 12
T2	the learning stage's events are queued from the stored labels	§4.6	test 12
T15	an erase reaches the record: span redacted, markers and window rebuilt	§4.4, §8.1	test 10
T16	the finding stage drops the redacted range from its record of what the window holds	§6.6	test 10
T22	a quarantined line already in the window is flagged and the window rebuilt	§4.4, §6.6	test 10

Test 8. Every marker resolves (§4.1, §5.1, §8.2). A property test over generated sessions, plus the nightly check; files read, then changed and deleted, must come back as read, and every one-line item's string must be in the entry it cites. Any failure is a defect.

Test 9. Trigger validity (§5.3, §6.5, §7.3). Slices against the provider's input count; evicted tokens by kind against the total; a thrashing session must trip the guard; a provider-tool session must show no runtime compaction. An unreconciled call breakdown keeps elective compaction off.

Test 10. Erase and quarantine (§4.4, §6.6, §8.1, Appendix D). Erase one span of a message that holds two corrections, and one string inside a file read, with reads issued before the epoch is applied; deliver the epoch twice, and kill the handler halfway. Quarantine an injected line already in the window, under each kind of controller; then pull an otherwise eligible item that the rebuild left out. Each of these is a defect:

- erased text served before the epoch is applied;
- a hit in the index, a marker or the controller's transcript;
- a lost neighbouring span, the rest of the file lost with the erased string, or a marker that stops resolving;
- a duplicate epoch that hangs or applies twice;
- a quarantined line in a call built after the receipt, or visible outside the audit view;
- a provider compaction reported as removal;
- a pulled item the finding stage still counts as in the window;

- a tool's or an API's copy neither removed nor named.

Test 11. Budget overflow and merges (§5.3, §5.5, §5.6). Run a session past the point where the owner's instructions exceed B_o, and on until the pointer lines alone would exceed it, then follow the directory past one page of D tokens; close the session and open the next. A defect: instructions taking more than B_o; a page over D; an instruction skipped, repeated or not recoverable word for word; a pinned instruction moved; a move not told to the owner and listed in a receipt; an earlier marker edited by a compaction that was not a merge; or markers from two sessions merged into one.

Test 12. Capture and adapter crashes (§4.6, §6.4, §6.6). One flush must capture a message holding two corrections. Kill the process before the append; after the durable append and before its acknowledgement (the retry must get the same address); after it and before the learning stage queues (the events must be rebuilt from the stored labels); between the hooks; and after injection, before delivery. Lag the tool's transcript. Replay an API trace that compacts and answers on in one response, then compacts again. A defect:

- a message reported as captured but not in the record;
- one message under two addresses;
- a marker shown as delivered but never seen;
- an unprotected inference missing from the receipt;
- a range the finding stage still counts as in the window after a compaction left its residency unknown.

Test 13. Never retained (§4.7). Plant one credential in an owner message, a file read and a controller injection; read one file in a temporary and an ordinary session, then end the temporary one; name another session's namespace in a read. A defect: the credential on a runtime disk, or unnamed in the tool's; anything findable after the temporary session ends; the ordinary session's blob gone; or the foreign read answered.

Test 14. Quoted authority, cuts and forged markers (§4.1, §5.1). The owner pastes the email of Table 4 and asks whether it is phishing; separately, one message holds two corrections in the owner's voice, one message is in a language the cutter has no word list for, and an unsure span is later confirmed. A tool result carries text shaped like a marker that quotes the "owner". A defect: the message not captured whole; the quoted line labelled, kept or passed on as an owner instruction; two corrections under one address; a stored span rewritten; the learning stage classifying a span again when it queues or replays it; or marker-shaped tool text read as a marker or as the owner's words.

Test 15. Waking a past session (§8.3). Wake a session after a planted supersede and after an erase of a span it read. A defect: an answer that uses the old value without saying it changed, erased text in the woken model's input, or a wake that leaves no later-visit entry.

Appendix F. The Price Rule in Full

This appendix holds the symbols, the cost of a pull and the worked example behind §7.2. Prices are multiples of the base input price, counted per call.

Table 7. The symbols of the price rule, in plain words, with the values of Figure 5.

Symbol	In plain words	Figure 5 value	Where it comes from
r	price of reading one cached token	0.1 (dashed: 0.05)	published; 0.05 on Opus and Sonnet 5.5
w	price of writing one token to the cache	1.25 (dash-dot: 2)	published; five-minute and one-hour cache
g	price of one output token	5	published ratio
S	tokens evicted	40K, 100K	chosen
m	tokens in the marker	300	assumed
A	tokens after the span	100K	assumed
L	context a pull rereads	150K	assumed
x	tokens a pull brings back	2K	assumed
o	output tokens of a pull	500	assumed
H	calls a pulled slice stays in the window	20	unmeasured
q_{call}	share of later calls that pull from the span	0 to 0.3 (x axis)	unmeasured
c	input price for the model that writes the marker	1	unmeasured

A pull is one more call. It rereads the L tokens of cached context, returns $x \ll S$ tokens that are written to the cache, and produces o output tokens, so $C_{\text{pull}} = rL + wx + g\,o$. The returned slice then stays in the window for H calls before it is evicted again, adding $r\,x\,H$ in later reads:

$$C_{\text{pull}}^{H} = C_{\text{pull}} + r\,x\,H = rL + wx + g\,o + r\,x\,H,$$

which is 24,000 units with the values above, and 20,000 without the slice's later reads.

Building the marker has a cost too. Exact strings and the owner's words need no model call (the strings are matched by code, the owner's spans were labelled at append), but the one-line items do. The model that writes them reads the span once at an input price c and writes the marker, so $K = cS + g\,m$, or 41,500 units for the 40K span. A merge reuses the one-line items already written (§5.4), so its K covers only entries no marker has covered yet; its rewrite term starts at the first merged marker.

The first term of N^{*}'s numerator is the finding paper's rewrite term (J2 §7.4): everything after the first changed token, here $m + A$ tokens, is written to the cache again at w instead of read at r, once per eviction. The denominator is the saving on each later call, $(S - m)\,r$, less the expected cost of pulls, $q_{\text{call}}\,C_{\text{pull}}^{H}$.

The worked example of §7.2 (a 40K-token span, five-minute cache) costs 156,845 units once: 115,345 for the rewrite and 41,500 for the marker. Each later call saves 3,970 units when nothing is pulled, so it pays after about 40 calls, 10 of them for the marker; at one pull in twenty calls the saving is 2,770 and break-even about 57; above about 0.165 it never pays. A 10K-token span never pays once more than 4% of later calls pull from it.

J-series paper J1 / Version 8.7 / 2026-10-21