The Authorship Floor

Why Project Knowledge Is Lost at Both Ends,

and What Anchoring It to Code Recovers

Akhil Katakam
Koragraph
katakamakhil0@gmail.com

September 2026

Abstract

Project documentation fails in two opposite directions at once. It loses what is too small to write down, the passing decision and the constraint that mattered for an afternoon, and it rots what is large enough to be worth writing, because nothing checks whether it is still true. We argue that these are one failure with one cause. Recording something in a document is a commitment to keep it true by hand, that commitment is expensive, and so developers record only what appears to justify the expense. The resulting threshold, which we call the authorship floor, produces both the rot above it and the silence below it, and neither half can be fixed alone: cheap capture without mechanical expiry yields a store that degrades as it grows, while automatic maintenance of documents recovers nothing that was never written down.

We describe a memory layer for coding agents built on the argument that both costs must fall together. A stored item is not a sentence but a claim bound, where it can be, to the program declaration it is about, resolved against a code graph at the moment of capture rather than recovered afterwards, and carrying enough of that declaration's content to recognise it again. Which items achieve that binding turns out to depend more on how they were captured than on what they say. When the code beneath an item changes, the item survives a rename or a move, is flagged when its subject is rewritten, or is withdrawn when its subject is gone. A fourth outcome carries as much of the design as the others: when the index is behind the working tree, or the file on disk disagrees with what was recorded, the layer returns no verdict and changes nothing, because absence of evidence is not evidence of falsity.

We report a worked lifecycle, two observations from running the layer on real work, and the limits of both. We claim no priority on the mechanism, which appears in work published shortly before this one; what we defend is the argument for why it is needed and an account of what such a system accumulates in practice. The system's source is public. This is a design paper: it argues a position and describes a working mechanism, and it does not report a controlled evaluation of whether that mechanism improves outcomes.

1 Introduction

Over an afternoon spent in an unfamiliar service, a developer learns three things. The retry wrapper swallows connection errors, so a test that asserts on them will never see one. The integration suite fails intermittently when run in parallel, and the failure has nothing to do with the code under test. And the connection pooling change should wait until the security audit lands next month.

A week later, at most one of these is written down anywhere.

The reason is not carelessness. Writing something into a project's instruction file, whether that is `CLAUDE.md`, `AGENTS.md`, `.cursorrules`, or the README, is an act of authorship, and authorship costs

a decision. The developer has to judge whether this particular thing is important enough to write down and, more consequentially, to keep true. The second half of that judgment is what does the damage. A fact recorded in a document is a maintenance obligation that nobody is assigned and nobody discharges, so the honest answer for most of what a developer learns is that it does not clear the bar.

We call the resulting threshold the *authorship floor*. It is not a formal construct. It is the implicit standard a developer applies before deciding to record something, and its effect is easy to observe in any repository: instruction files contain architecture summaries, house style, and a handful of hard-won warnings, and they contain almost nothing that was true for only a week.

The floor produces two failures at once, and they pull in opposite directions.

Below the floor, knowledge is never recorded at all. The three facts above are worth having and none of them is worth a paragraph in a maintained document. They are specific, they are timely, and they expire on their own. The pooling constraint is irrelevant once the audit finishes. The flaky test may be fixed next sprint. A developer who wrote each of these into a document would be committing to prune it later, which is precisely the work nobody does, so they write none of it and the knowledge leaves with the session that produced it.

Above the floor, knowledge is recorded and then rots. The warning about the retry wrapper is worth writing, so it gets written, naming the wrapper by name. Some months later the wrapper is renamed during a refactor. The sentence in the instruction file does not change, because nothing connects the two. It now names a function that does not exist, and it is still delivered on every turn, as an instruction, to every agent and every new engineer who reads the file. Nothing in the system knows.

This is not a hypothetical. A study of 247,694 instruction lifetimes across 1,867 repositories finds that agent instruction files more than triple over their lifetime, gaining close to five net instructions per commit, and that the probability of an instruction being deleted *falls* the longer it has been there [5]. Instructions accumulate, and the old ones, which are the likeliest to have gone stale, are the ones least likely to be removed.

These look like different problems and they are usually treated as different problems. Documentation rot has a literature; the loss of small, situated knowledge mostly does not. We argue that they are one problem with one cause. Authorship is expensive, and maintenance is manual. The first cost sets the floor, and the second is why the floor has to be set so high: a developer records only what they are willing to keep true, and keeping things true by hand is expensive enough that the willing set is small.

The consequence is that the two costs cannot usefully be attacked separately. Reducing the cost of writing something down, on its own, produces a store that accumulates without bound and degrades as it grows. Making documents easier to maintain, on its own, does nothing for the knowledge that was never written because it was too small to bother with. Both costs have to go together, and the mechanism that removes the second is what makes it safe to remove the first.

This paper describes a memory layer for coding agents built on that argument. Capture is incidental, derived for the most part from a session's own event stream rather than authored. Expiry is mechanical, because a stored item is not a sentence but a claim bound to a specific declaration in the code, checked against that declaration whenever the repository is indexed. When the code beneath an item changes, the item survives, is flagged for review, or is dropped, and when the system cannot tell, it says so and changes nothing.

We make three contributions. We name and argue for the authorship floor as the common cause of two documentation failures usually treated separately, and we argue that the two costs it comprises cannot be attacked one at a time. We describe a working design in which memory validity is a decidable property

of the repository rather than an assumption, including its treatment of the case where validity cannot currently be determined. And we report two observations from running it on real work, one of which is that where an item enters such a system predicts whether it can ever be checked better than what the item says.

We claim no priority on the mechanism itself. Anchoring a claim to the artifact that established it, and declining to guess when the anchor cannot be resolved, both appear in work published a month before this preprint [13], which we came across while preparing it and did not build on. What we offer alongside the mechanism is the argument for why it is needed, which is a claim about documentation rather than about any implementation of it, and an account of what such a system accumulates when pointed at real work.

This is an early exploration rather than a finished account. It argues a position, describes a system built to act on that position, and reports what we have seen so far, which is less than we would like to report and more than we expected to be surprised by. It does not contain a controlled evaluation. The source is public, and readers can run it against their own repositories.

2 Why Both Costs Have to Fall Together

The argument of this paper is that the authorship floor cannot be lowered by attacking either of its two costs alone. It is worth being explicit about why, because each half looks sufficient on its own and neither is.

2.1 Cheap capture, by itself, makes things worse

Suppose the cost of writing something down goes to zero. An agent records every constraint it is told, every failure it recovers from, and every decision made in passing. The store fills quickly, which is the intent, and then it keeps filling, which is the problem.

Nothing in such a store distinguishes a constraint that expired last month from one that still holds. The pooling deferral from our opening example is indistinguishable, as text, from a rule that will be true for the life of the project. Both are sentences. Both are retrieved. A reader receives them together and has no basis for discounting either.

This is not merely untidy. The material a store delivers is delivered into a budget that is shared with the task at hand, and adherence to instructions degrades measurably as the number of instructions grows [15]. Stale material therefore does active harm twice over: it consumes budget that correct material would otherwise occupy, and it dilutes the instructions that remain. A store that has accumulated for six months without pruning can be worse than no store at all, because an empty store at least leaves the budget intact and does not assert anything false.

The usual response is to forget on a schedule. Discard the least recently used, decay by age, or periodically ask a model to summarise and compress. Every one of these approaches is guessing, and each guesses badly in the same direction: rules that are rare and permanent look exactly like rules that are rare and obsolete. A house convention consulted twice a year is the first thing an importance heuristic discards, and it is precisely the kind of knowledge that was worth recording.

2.2 Cheap maintenance, by itself, recovers nothing

Now suppose instead that documents maintain themselves. Some mechanism watches the repository, finds the sentences that no longer hold, and flags or removes them. Instruction files stay accurate.

This solves the failure above the floor and leaves the floor exactly where it was. The pooling deferral was never in a document, so no document maintenance mechanism will ever see it. The flaky test was never written down. The retry wrapper's behaviour was recorded only because somebody judged it worth the trouble, and that judgment is what the floor consists of.

Perfect maintenance of documents that were never written is worth nothing. Whatever fraction of a project's working knowledge sits below the floor stays lost, and our contention is that this fraction is large.

2.3 Forgetting is what licenses remembering

Put the two halves together and each supplies what the other lacks.

Mechanical expiry is the precondition that makes cheap capture safe. If an item can be checked against the code it describes, and removed when that code is gone, then recording it costs nothing that has to be repaid later. The maintenance obligation that the authorship floor exists to avoid is discharged by the system rather than by the developer, and the judgment that produced the floor is no longer required. A developer, or an agent, can record something true for a week without thereby committing to anything.

The relationship runs in one direction. Capture is the feature; expiry is what makes it permissible. A design that offers only the first is a store that degrades. A design that offers only the second is a document linter. We state the principle in the form we find useful when reasoning about such systems: you can only afford to remember small things if forgetting is automatic.

What remains is to make forgetting mechanical, which requires a stored item to be something more than a sentence.

3 Anchored Memory

Making forgetting mechanical requires a stored item to be something more than a sentence. This section describes what we store instead, and what can be decided about it.

3.1 A claim and a referent

An item consists of a claim, written in ordinary language, together with a referent: the repository, file, and declaration the claim is about. The referent is not a citation in the text. It is a row resolved against a code graph at the moment of capture, and it carries enough of the declaration's content to recognise that declaration again later.

Three of the referent's fields do the work. The *grain* records whether the claim is about a single declaration, a file, or the repository as a whole. The *body fingerprint* is a truncated SHA-256 of the declaration's normalised body, which answers whether the code is byte-identical to what was seen. The *body sketch* is a set of hashed three-token shingles over the same body, truncated to sixty-four values, which answers a different question: whether some declaration is the same code under a different name.

Two signatures are needed because a rename defeats the first. The declaration line is inside the body being hashed, so renaming a function changes its fingerprint completely, and an exact hash cannot distinguish that from the function having been deleted and an unrelated one written in its place.

The two sketches are compared with a bottom-k Jaccard estimator computed over their union. The naive alternative, intersecting two independently truncated lists, undercounts whenever either body exceeds the sketch size: the smallest k hashes of each body come from different windows of the hash space. Shingles are over tokens instead of lines because short declarations are the common case. A four-line

function yields two line-shingles, a rename destroys one of them, and the similarity that results is indistinguishable from noise.

Re-resolution after a rename requires two conditions rather than one. The best candidate must clear a similarity floor, and it must also beat the runner-up by a margin. A single threshold is not enough in a repository containing families of near-identical declarations, where several candidates clear any floor low enough to be useful and picking the highest is close to picking arbitrarily. When no candidate dominates, the layer refuses to re-resolve rather than guessing.

3.2 What can be decided

Every time a repository is indexed, each anchor is checked against the current state of the code. The check returns one of six verdicts.

Three of them mean the claim's subject is still there. `ok` is an unchanged declaration. `renamed` and `moved` are declarations found again under a new name or in a new file, in which case the anchor is rewritten to point at the new location and the previous name is retained alongside it, so the history of the binding is not lost.

One means the subject changed underneath the claim. When a declaration's body no longer matches its fingerprint, the verdict is `unconfirmed`. The layer records that the claim needs a look and deliberately declines to decide anything further. Whether a rewritten body falsifies a claim about it is a judgment this mechanism is not equipped to make: a rationale can survive a complete reimplementation, and a description of behaviour can be falsified by a two-line change. Treating drift as falsification would discard true claims at exactly the moment they were most likely to be relevant, and so it is treated as a flag rather than a verdict against the claim.

One means the subject is gone. `orphaned` is returned when the declaration cannot be found anywhere in the branch. It is the only verdict that withdraws an item, and it does so only when every anchor on that item is orphaned. Withdrawal by this route is not the only way an item leaves delivery, and it is not the most common one. A later statement of the same rule supersedes an earlier one, and a developer can retire an item by hand. Both happen more often than orphaning. An item carrying several anchors survives while any one of them still resolves, and an item with no anchors at all is never expired by this path.

Withdrawal is not deletion. Nothing in the layer removes an item from the store. Expiry writes a timestamp and a reason, the row and its anchors remain, and the store can therefore answer why it stopped delivering something, which a store that deletes cannot.

3.3 The verdict that changes nothing

The sixth verdict is `unknown`, and it carries more of the design's weight than the other five.

The code graph is a separate artifact from the working tree and is maintained asynchronously. It is routinely behind. A developer edits a file, the graph has not been rebuilt, and a symbol that exists on disk is absent from the index. A system that reads that absence as deletion will withdraw a true claim, and it will do so most often during active editing, which is when the claim was most likely to matter.

So the layer distinguishes not finding something from having established that it is gone. It returns `unknown` when there is no graph, when the repository is not in it, when the checkout's location is unknown, when a file is on disk but not yet indexed or no longer indexed, when the indexed commit disagrees with the checkout's current commit, when the file's recorded content hash disagrees with the file on disk, when a re-resolution target is ambiguous, when several declarations share the name and none matches by

fingerprint, and when a body cannot be read. Nine reasons in all. None of them expires anything.

The divergence checks are written so that uncertainty resolves toward silence. Divergence is established only when both the indexed commit and the checkout's commit are known and differ. Either one being unavailable is not divergence, it is another `unknown`. The same holds at file granularity, where the content hash recorded at indexing time is compared against the file on disk and a mismatch downgrades any orphan verdict rather than confirming it.

The principle is that absence of evidence is not evidence of falsity. Every mechanism above exists to keep the layer from asserting something it has not established, and the cost is accepted knowingly: a system that says nothing when it cannot tell will sometimes carry a stale item longer than necessary, which is a smaller harm than deleting a true one.

Two limits follow from all of this. Repository-grain anchors have no declaration to resolve, so the resolver above returns `ok` for them without checking anything. A second and much narrower mechanism covers part of that gap: a rule naming a tool preference, of the form use this and not that, leaves structural evidence in the project's manifests, and that evidence is checked. A disagreement is recorded and the item is withheld from delivery rather than expired, on the ground that a manifest is weaker evidence than a person and a developer should settle the conflict. It fires rarely. The wider limit is that what any of this decides is existence, not truth. Section 7 takes both up.

4 Capture Without Authorship

Mechanical expiry removes the maintenance cost. It does not, on its own, remove the cost of deciding to write something down, and a system that still requires a developer to author each item has moved the floor rather than dissolved it. The layer therefore derives items from a session's own activity as well as accepting stated ones.

How much it derives in practice is a separate question from whether it can, and the answer depends on how the layer is used. A project that imports an existing instruction file and states rules by hand will hold mostly authored items whatever the derivation machinery is capable of. Section 7 returns to this, because it bears on whether the floor is lowered in practice or only in principle.

4.1 Distillation from a session's events

A hook records what a coding session does: which files were read and edited, which commands ran, which failed, and what they printed. These events are not memory. They are a stream from which memory can be distilled offline, which is deliberate, because a distillation rule that runs inside the hook cannot be improved without losing the data captured under the old rule.

The obvious distillation rule is that a failure followed by a later success on the same command is a lesson. In practice this rule extracts almost nothing, and the reason is specific to agents. A human developer runs the same test command repeatedly, so command identity is a reliable key. An agent writes a new heredoc, a new inline script, and a new scratch path on nearly every attempt, so successive invocations that a human would consider the same command share no text at all. Keying on command identity therefore matches so rarely as to be useless.

What replaces it is the disappearance of an error signature. The layer arms on a failure whose output is about the code rather than about the environment, requires at least one intervening edit, and then accepts a pass from a command of the same *shape* as evidence that the signature is gone. Shape is what survives normalising an invocation: the executable and its subcommand, with heredocs, temporary filenames and

inline scripts stripped away. A dozen textually distinct commands that all run the test suite share one shape. Classifying the failure comes first and matters more than it appears to: a command killed by a timeout prints real assertion failures on its way to being killed, so environmental causes are checked before the failure is read as evidence about the code at all. We report the substitution, from command identity to error signature and command shape, as a small transferable result for anyone mining agent traces.

4.2 What was tried and abandoned

The lesson above records the edit that finally worked. The hours that preceded it went into approaches that were tried and backed out, and those are what stop a later session from beginning at the first of them again. Nothing else preserves them. Version control does not, because the approaches were never committed, and the working tree does not, because they were removed.

Three structural signals produce such items, and each is mechanical rather than inferred. The first is a revert, detected when content that a scope had already removed reappears, in which case the approach standing at the moment it reappeared is the one that was discarded. The second is a command shape that failed repeatedly under one error signature and never subsequently passed, where a single failure is a mistake and repetition is a statement about the repository. The third is a file edited and then reverted within a single turn.

4.3 Three tiers, one of which is never delivered

Items differ in how much authority they can claim, and the layer records this rather than flattening it. An item is a *law* when the developer stated it, an *observation* when it was derived mechanically and carries its provenance, and a *hypothesis* when something was inferred.

Hypotheses are stored and never delivered. No read path returns them, so an inference that the layer was not confident enough to promote cannot reach a reader through any surface. This is the consequence of a position worth stating plainly: a memory system for an agent has an obligation to refuse before it has an obligation to rank, because a confidently delivered wrong memory is worse than silence, and inference is where wrongness comes from. A meaningful fraction of what the layer has captured has never been shown to anyone.

We note one thing about how that property is enforced, because it bears on how such properties should be built. The filter is applied in the query rather than left to callers, which is the right choice, but it is applied in every query rather than at one point, and during the preparation of this paper we found a read path that had been added without it. A drifted hypothesis could reach a reader through that path. It has been fixed, and the episode is worth reporting rather than concealing: a property enforced by a predicate repeated across many queries is a property that will eventually be missed in one of them, and stating the invariant in prose is not the same as having a single place where it is imposed.

5 A Worked Lifecycle

The following sequence was run against a repository created for the purpose. It is an illustration of the mechanism rather than an evaluation of it, and we describe it in that spirit: it shows what the design does, not how often it helps.

A small module defines two functions, one of which computes a running balance from a list of ledger entries. A developer states a rule about it in the course of ordinary work, naming no coordinates and passing no arguments, simply writing that the opening balance handed to `reconcileLedger` must already

be in cents.

The layer extracts candidate referents from that sentence. Most of its words are excluded by shape: a candidate must be at least four characters and must carry a hump or an underscore, so *opening*, *balance*, *handed* and *cents* are never probed. One word survives, it resolves to exactly one declaration in the graph, and the item is bound to that declaration at symbol grain with a fingerprint and a sketch of its body. Had two words resolved, or had one word resolved to two declarations, the item would have fallen back to the repository as a whole, since nothing in the sentence would then identify which declaration it was about.

The function is renamed to `settleLedger` and the repository is re-indexed. The fingerprint no longer matches, because the declaration line is inside the hashed body, but the sketch of the new declaration is close to the stored one and no competing candidate is close enough to contest it. The verdict is `renamed`. The anchor is rewritten to the new name, the previous name is retained beside it, and the item stays live and continues to be delivered, now pointing at code that exists.

The function is then deleted outright and the repository re-indexed again. No declaration in the branch resembles the stored sketch. The verdict is `orphaned`, it is the item's only anchor, and the item is expired with that reason recorded. It remains in the store. Asked about it afterwards, the layer reports the claim, the declaration it was about, the rename it survived, and the reason it was withdrawn.

The same rule written into an instruction file would have passed through all three of these states unchanged. After the rename it would have named a function that no longer existed while continuing to be delivered on every turn, and after the deletion it would have done so with no remaining referent at all. Nothing in that arrangement is capable of noticing, because a sentence in a document has no relationship to the code it describes beyond the reader's willingness to check.

We draw attention to what the sequence does not show. It does not show that the rule was useful, or that any agent behaved differently for having received it. It shows that the binding was formed without anyone being asked to form it, that it survived a change which would have broken a textual reference, and that it was withdrawn when its subject ceased to exist. Those are properties of the mechanism, and they are what the design claims.

6 Observations From Use

We have been running the layer on our own work while building it. That makes a census of the store close to meaningless: aggregate counts over such a window describe the construction of the system as much as the use of it, and items entered in bulk during setup sit alongside items captured during ordinary work with nothing in the totals to separate them. We therefore report two observations that are structural rather than temporal, and we do not report totals.

6.1 Where an item lands is decided by how it was captured

The first observation is visible in Table 1, and it holds regardless of when any particular item was created, because it concerns which grain a capture path is capable of reaching rather than how many items passed through it.

History mining reaches a declaration almost always, because the unit it mines is a commit hunk and a hunk sits inside one. A rule the developer states reaches a declaration rarely, because it resolves one only when its sentence happens to name something the graph knows, and people state policy more often than they state facts about particular functions. Items read out of existing instruction files behave like stated rules, which is unsurprising, since that is what they are.

Capture path	Symbol	File	Repository
Mined from repository history	73	9	0
Distilled from session events	6	47	0
Imported from instruction files	2	220	129
Stated by the developer	4	36	133

Table 1: Live anchors by capture path and grain. Only symbol-grain anchors are subject to the re-resolution described in Section 3.

This is worth drawing out because it cuts against the intuition the design encourages. One expects the limiting factor on whether a claim can be checked to be the claim: some things are about declarations and some are not. What the table suggests is that the capture path matters at least as much. The same knowledge, arriving through a hunk rather than through a sentence, binds to code and becomes checkable, and arriving through a sentence, does not.

6.2 Most sessions have nothing to teach

The second observation concerns what a session produces. Of 537 recorded episodes, an episode being one prompt and the activity it produced, 398 were read-only or ended with no edit at all. Only 134 changed the working tree.

A session that meets no friction leaves nothing worth remembering, which is obvious once stated and has a consequence that is not. Any attempt to measure a memory layer by sampling development tasks uniformly will spend most of its budget observing sessions with nothing to capture. Tasks have to be selected for friction, or the measurement is largely of nothing happening.

6.3 What we can and cannot say yet

The mechanisms described in Sections 3 and 4 do fire. Renames have been followed, drift has been flagged and left for review, tool-preference rules contradicted by a manifest have been recorded and withheld, and items whose declarations were deleted have been withdrawn. We are not reporting rates for any of these, because the window that produced them includes the period in which the resolver itself was being written, and a rate computed across that boundary would describe our development more than the design.

Nor does anything here show that the layer improved an outcome. It shows that the mechanism runs on real work and that two things about it are worth noticing. Whether it helps is a different question, and this paper does not answer it.

7 Limits and Open Questions

What follows is the edge of the design: the places where the mechanism stops working, and the questions we would want answered before claiming more for it. Some of these are limits we expect to be permanent, and some are open problems we think are the interesting part of the idea.

Existence is weaker than truth. What the mechanism decides is whether a declaration still exists, and that is a narrower thing than whether a claim is still true. A claim about the relationship between two declarations can be falsified by an edit to a third. A claim about behaviour behind a stable interface can be falsified without the interface changing. A claim about the world outside the repository, a rate limit or an

upstream contract, has no referent in the code at all. Decidability is bought for one subclass of claims and the layer says which claims those are, which we think is the right trade, but the boundary of that subclass is where we would look first for a stronger mechanism.

Policy has nothing to bind to. A rule stated as policy about a project names no declaration, and the declaration resolver never checks it. The manifest check of Section 3 covers one narrow shape of such a rule, where a tool preference leaves structural evidence somewhere, and the rest is carried without any test.

This is the sharpest open question the design raises, and it complicates the argument of Section 2 in a way worth stating. Lowering the cost of recording something does not by itself make it checkable. A rule about how a project is worked on may need a different kind of validity mechanism rather than a weaker version of this one, and we suspect the answer involves finding the structural trace that a policy leaves somewhere in the repository, as the manifest check does for one case. What that generalises to, we do not know.

A rewrite is flagged, not judged. When a body changes underneath a claim the layer records that the claim needs a look and stops there. We think the refusal is right, because both possible errors are costly and neither is visible in the fingerprint, but it does mean drift accumulates as a queue rather than resolving itself. Whether such a queue is reviewed in practice, or quietly grows until it is ignored, is a property of long-running use that we have not run long enough to observe. There is also a live possibility that a classifier could answer what we decline to answer, in which case part of our refusal would be a limitation wearing the clothes of a principle.

Referent extraction is a guess. Binding a claim to a declaration from a sentence means guessing which words name code. The implementation guesses from shape, which misses lowercase single-word declarations by construction and will sometimes propose a word that merely looks like an identifier. Where the guess fails the claim falls back to a coarser grain rather than binding wrongly, so failures cost coverage instead of correctness. How much coverage they cost is unmeasured and would be straightforward to measure, and Table 1 suggests it is the single lever with the most room in it: a better extractor moves items from the rows that cannot be checked into the row that can.

One user, who wrote the system. The observations in Section 6 come from one developer, on repositories we wrote, using a system we wrote, while we were writing it. The store may look as it does because the design is right, or because we state rules in the shape our own extractor happens to parse, and nothing here separates those. They are an existence proof that the mechanism runs on real work and two things we noticed while it did.

Usefulness is untested. We report what is captured and what becomes of it, not whether it helped. Whether an agent that receives such items works better than one that does not is the question a reader will want answered, and answering it requires a design this paper does not attempt: tasks selected for friction, a second encounter with the same difficulty, and a measure that survives the observation in Section 6 that most sessions have nothing to teach. That is the experiment we would run next.

8 Related Work

Recovering links after the fact. Antoniol et al. [2] fixed the shape of the problem the field has worked on since: given code and documents written independently, recover the links post hoc with information retrieval over document terms, identifiers and comments, at a quality bounded by lexical overlap between artifacts nobody wrote with linking in mind. Our anchors are not recovered. They are resolved when the claim is written, against a code graph that already knows which declaration a name refers to, so binding is a lookup rather than an inference. That is smaller than it sounds: we cannot link an item to code it never mentions, the case IR recovery exists to serve.

Suspect links and event-based traceability. Cleland-Huang, Chang and Christensen [7] had artifacts publish change events to subscribers, so a change at one end marked the dependent links suspect and queued them for review. We concede the descent: our verdict lattice is a suspect-link scheme with more resolution, and *unconfirmed* is a suspect link renamed. Mäder and Gotel [17] recognised development activities from a stream of elementary changes in a modelling tool, then applied rules that update the affected links automatically. Hammad, Collard and Maletic [12] judged from the code side, using syntactic differencing to decide whether a source change actually alters the class diagram. Rahimi and Cleland-Huang [22] combined refactoring detection with IR heuristics to evolve requirements-to-code links across contiguous versions. Each tries to decide whether a link is still correct, treating inability to decide as a gap to close. We make that inability a verdict: *unknown* is returned under nine conditions, none of which expires anything, and divergence is established only on positive evidence that two commits are both known and different. Their tools needed it less, sitting inside a managed repository and seeing every change to both endpoints; our index is a cache that may sit arbitrarily behind the working tree.

Links created by interaction rather than recovery. Hübner and Paech [14] created requirements-to-code trace links continuously from developer interaction logs in the IDE, avoiding retrospective recovery and keeping links usable during a project rather than after it. This is the closest published precedent for binding links as work happens rather than recovering them later, and we claim nothing over it. Two differences survive. IDE interaction is a noisy signal for relevance, since a developer opens a file for many reasons, whereas an agent's tool call names the declaration it is reasoning about. And their endpoints are requirements some process already obliged someone to write; ours are claims no tracker would hold.

Code and comment consistency. Tan et al. [27] extracted implicit rules from comments in four large C systems and checked code against them. Ratol and Robillard [24] detected comments made fragile by an identifier rename, at the moment of the rename. Panthaplackel et al. learned to rewrite a comment from the code change that outdated it [19], and to classify before a commit lands whether a change has made its comment inconsistent [20]. Wen et al. [29] mined 1.3 billion AST-level changes across 1,500 systems for a taxonomy of the inconsistencies developers fix; Aghajani et al. [1] catalogued documentation problems as practitioners report them. This literature decides what our layer refuses to decide: a just-in-time classifier answers the question *unconfirmed* declines, namely whether a rewrite falsifies a sentence. If such a classifier is accurate enough in practice, part of our refusal is a limitation dressed as a principle, and we would say so. Our narrow defence is that those models are trained on pairs where the comment describes the method, and much of what we capture describes nothing (a deferral, a note that a test is flaky under `-j`). The errors are asymmetric too: an item wrongly kept is noise, one wrongly expired leaves silently.

Entity identity across versions. Godfrey and Zou [11] used origin analysis to determine where a function or file came from when entities merge and split. RefactoringMiner [28] detects refactorings in commit histories without similarity thresholds, at the best reported precision and recall; RefDiff [25] reaches comparable ends with static analysis and similarity heuristics across several languages. We concede these subsume our identity mechanism and do it far better. A bottom-k sketch over 3-token shingles cannot distinguish an extracted method from a rewrite, and a declaration split in two presents to us as *orphaned* or *unconfirmed*, never as split. The sketch was chosen for cost and language reach on a developer machine, and policy partly rescues it: only total absence of every anchor withdraws an item, so a mislabelled rename changes the verdict text, not what the reader receives. That argues the weakness is survivable, not that it is not a weakness.

Agent memory systems. MemGPT [18] manages a virtual context, paging items between the window and external storage. Mem0 [6] extracts and consolidates facts from conversation for retrieval. A-MEM [30] builds Zettelkasten-style notes with generated attributes and links between them, and Generative Agents [21] retrieve from a memory stream weighted by recency, importance and relevance. In all of them a stored item is text retrieved by similarity, with no referent outside the store, so none can consult what a memory is about; removal comes from a later contradicting item or from decay. Zep [23] goes furthest: its Graphiti engine maintains a temporal knowledge graph, uses a model to judge whether a new edge contradicts existing ones, and stamps the superseded edge with validity bounds instead of deleting it. Our non-destructive expiry is the same instinct and we claim no priority for it; the difference is where the judgement comes from. Zep invalidates from later text through a model, while we compute from the artifact a claim was bound to and decline when it cannot be read.

Concurrent work anchoring claims to artifacts. EA-Graph [13] anchors a coding agent's verification claims to the artifact content used to establish them, resolves aliases to leaf definitions, separates evidence strength from freshness, and returns a claim as unprovable rather than guessing when replacement content is unavailable. It reaches both anchoring at capture and the refusal to guess when an anchor cannot be resolved, independently and about a month before this preprint, and we claim no priority over either. The defensible distinction is scope: EA-Graph carries verification claims evaluated on generated repositories with ground truth known by construction, while our items are arbitrary developer knowledge captured incidentally from real sessions.

Beliefs held below a delivery threshold. NELL [4] accumulated candidate beliefs continuously and promoted them into its knowledge base only once evidence from several extractors crossed a threshold, leaving the rest unpromoted in the system. This partially anticipates our hypothesis tier, and the anticipation should be granted: a learning store that keeps material it does not expose is not our idea. Our tier differs only in what sets it, since provenance fixes it at capture, by whether a developer stated the claim or a mechanism derived it, rather than confidence rising to a threshold.

Truth maintenance and time in stored data. Doyle's truth maintenance system [9] records justifications for beliefs and propagates retraction through them; de Kleer's assumption-based variant [8] labels each datum with the assumption sets under which it holds. The vocabulary invites an overclaim, so we state it flatly: koragraph is not a truth maintenance system. It records tiers and contradiction edges but no justification structure, and withdrawing an item propagates to nothing. Snodgrass and Ahn [26] separated valid time, when a fact holds in the world, from transaction time, when the database learned it. Our expiry

timestamp is transaction time alone: the moment the layer observed an anchor's subject gone, not the moment the claim stopped being true.

Second indexes over source code. Kythe [16] and Glean [10] store schema-defined facts about declarations, references, types and calls, serving cross-references and code search at scale. CodeQL, whose query language is described by Avgustinov et al. [3], compiles object-oriented queries to Datalog over a relational encoding of a program. A second index over the repository is their pitch as much as ours; our graph is a smaller, less rigorous version of it. The payload is the difference. What they hold is derived from code and recomputable from it, which makes staleness there a performance question. A natural language claim is recomputable from nothing, so it must be maintained rather than regenerated.

Why accumulation is not free. A store that only grows moves its cost to read time. Jaroslawicz et al. [15] measured instruction-following against instruction density and found adherence degrading as the count rises, the strongest models reaching 68 per cent accuracy at 500 instructions. Chakrabarti [5], whose figures we gave in Section 1, observed the accumulation itself: files that triple in length while their oldest instructions become the hardest to remove. Together they are the empirical form of this paper's argument: cheap capture with no mechanism that removes an item when its subject is gone produces a store whose value per item falls as it grows.

9 Conclusion

Documentation is usually discussed as though its problem were accuracy. Accuracy is the visible half of a problem whose other half leaves no trace: the knowledge never written down produces no artifact to be found wrong later, so it is absent from the discussion in the way that anything unrecorded is absent. Both halves come from one place. Recording something is a commitment to keep it true by hand, that commitment is expensive, and so people record only what seems to justify the expense. The threshold this produces is what we have called the authorship floor.

Removing either cost alone does not work, which is the argument this paper is built on. Cheap capture without expiry produces a store that grows until it costs more than it returns, and there is now direct evidence of exactly that happening in agent instruction files. Automatic maintenance without cheap capture keeps documents accurate and recovers nothing that was never in a document. The two have to fall together, and what makes that possible is that the mechanism which makes forgetting automatic is the same one that makes remembering safe.

Binding a claim to the declaration it describes is one way to get there, and it is cheap to form only because an agent can form it as a side effect of work it was doing anyway. A system built this way can tell that a claim's subject was renamed, or rewritten, or deleted, or that it currently cannot tell at all. That last state is the part we would most encourage others to take, whatever they make of the rest: a memory system that answers when it has no basis to answer is not being helpful, and admitting uncertainty costs far less than being confidently wrong to something that will act on the answer.

What we have built is an early version of an idea rather than a finished account of it. The design works, in the narrow sense that the mechanism runs and does what Section 3 says it does, and Section 7 is honest about where it stops. The parts we find most interesting are the ones we cannot yet answer: what validity means for knowledge that names no code, how much coverage a better referent extractor would buy, and whether any of this changes how the work goes. Those are the questions we would want a reader to take from this, more than the implementation that prompted them.

Artifact

The system described here is at `https://github.com/Koragraph/KoragraphMCP`. Its source is public under the Business Source License, which permits reading, modification and non-production use, and is not an OSI-approved open source licence. All behaviour described in Sections 3 through 5 was verified by reading that source.

References

[1] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza. Software Documentation Issues Unveiled. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1199-1210, 2019.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970-983, 2002.

[3] P. Avgustinov, O. de Moor, M. Peyton Jones, and M. Schäfer. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP)*, LIPIcs volume 56, pages 2:1-2:25, 2016.

[4] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, and T. M. Mitchell. Toward an Architecture for Never-Ending Language Learning. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 1306-1313, 2010.

[5] K. Chakrabarti. Why Does CLAUDE.md Keep Growing? Catastrophic Remembering in Agentic Coding. arXiv preprint arXiv:2608.11095, 2026.

[6] P. Chhikara, D. Khant, S. Aryan, T. Singh, and D. Yadav. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. arXiv preprint arXiv:2504.19413, 2025.

[7] J. Cleland-Huang, C. K. Chang, and M. J. Christensen. Event-Based Traceability for Managing Evolutionary Change. *IEEE Transactions on Software Engineering*, 29(9):796-810, 2003.

[8] J. de Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127-162, 1986.

[9] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231-272, 1979.

[10] Meta Open Source. Glean: A System for Collecting, Deriving and Working with Facts about Source Code. `https://glean.software/`. Accessed 2026.

[11] M. W. Godfrey and L. Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering*, 31(2):166-181, 2005.

[12] M. Hammad, M. L. Collard, and J. I. Maletic. Automatically Identifying Changes that Impact Code-to-Design Traceability During Evolution. *Software Quality Journal*, 19(1):35-64, 2011.

[13] H.-J. Hsu, C.-J. Chi, and H. Everett. EA-Graph: Artifact-Anchored Verification Memory for Coding Agents under Upstream Drift. arXiv preprint arXiv:2608.04278, 2026.

[14] P. Hübner and B. Paech. Interaction-Based Creation and Maintenance of Continuously Usable Trace Links between Requirements and Source Code. *Empirical Software Engineering*, 25(5):4350-4377, 2020.

[15] D. Jaroslawicz, B. Whiting, P. Shah, and K. Maamari. How Many Instructions Can LLMs Follow at Once? arXiv preprint arXiv:2507.11538, 2025.

[16] The Kythe Project. Kythe: A Pluggable, (Mostly) Language-Agnostic Ecosystem for Building Tools that Work with Code. `https://kythe.io/`. Accessed 2026.

[17] P. Mäder and O. Gotel. Towards Automated Traceability Maintenance. *Journal of Systems and Software*, 85(10):2205-2227, 2012.

[18] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. arXiv preprint arXiv:2310.08560, 2023.

[19] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney. Learning to Update Natural Language Comments Based on Code Changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1853-1868, 2020.

[20] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney. Deep Just-In-Time Inconsistency Detection Between Comments and Source Code. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, volume 35, number 1, pages 427-435, 2021.

[21] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 2:1-2:22, 2023.

[22] M. Rahimi and J. Cleland-Huang. Evolving Software Trace Links between Requirements and Source Code. *Empirical Software Engineering*, 23(4):2198-2231, 2018.

[23] P. Rasmussen, P. Paliychuk, T. Beauvais, J. Ryan, and D. Chalef. Zep: A Temporal Knowledge Graph Architecture for Agent Memory. arXiv preprint arXiv:2501.13956, 2025.

[24] I. K. Ratol and M. P. Robillard. Detecting Fragile Comments. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 112-122, 2017.

[25] D. Silva and M. T. Valente. RefDiff: Detecting Refactorings in Version Histories. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 269-279, 2017.

[26] R. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, 19(9):35-42, 1986.

[27] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments? */ In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 145-158, 2007.

[28] N. Tsantalis, A. Ketkar, and D. Dig. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, 48(3):930-950, 2022.

[29] F. Wen, C. Nagy, G. Bavota, and M. Lanza. A Large-Scale Empirical Study on Code-Comment Inconsistencies. In *Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 53-64, 2019.

[30] W. Xu, K. Mei, H. Gao, J. Tan, Z. Liang, and Y. Zhang. A-MEM: Agentic Memory for LLM Agents. arXiv preprint arXiv:2502.12110, 2025.