Capture at the Write Boundary: Hook-Driven Decision Recording in Agentic Development

kgai maintainers

kgai.dev · team@kgai.dev · August 2026

ABSTRACT

Software teams lose the rationale behind structural decisions because documentation is written after the fact, when the reasoning has already begun to fade. We describe the capture model of kgai, a local-first, immutable knowledge graph of engineering decisions, which records at the write boundary: the moment within an AI-assisted coding session when a structural change is made and its rationale is still in the model's working context. Capture rests on three instruments. A session-start hook injects the project's capture rules, a skill directs the model to record structural decisions as part of completing the task, and a stop hook intercepts sessions that edited code without recording. We define what qualifies as a decision (a structural, domain-level choice) and what does not (code-level renames, formatting, unacted analyses). Because the write path is a standalone command-line interface, capture binds to the language model through instructions rather than to a particular agent harness. We state the limitations plainly: edits outside the instrumented loop are not captured, and manual recording or import remains the fallback.

1. Introduction

Every codebase embodies decisions whose reasoning is rarely written down. Empirical studies of developer work habits show that engineers keep this knowledge in personal mental models and recover it by reading code and interrupting colleagues, turning to written repositories of design knowledge only reluctantly [1]. Where documentation exists, practitioners rate it as valuable in principle but persistently out of date in practice [2]. Architecture decision records were proposed as a lightweight countermeasure, one short file per decision kept in the repository [3], and the practice has spread widely. Yet decision records share the weakness of all retrospective documentation. A person must notice that a decision has been made, stop, and write.

Agentic development changes the setting in one specific way. When a language model performs the change, the rationale is present in the model's working context at the moment the change is written, in a form the model can articulate on request. That rationale is also uniquely perishable, because the context is discarded when the session ends. We call the moment at which a structural change is written, while its

rationale is still in context, the write boundary, and we argue that it is the correct place to capture engineering decisions.

This report describes how kgai, a local-first, immutable knowledge graph of engineering decisions, instruments the write boundary of AI-assisted coding sessions. We describe where capture happens, what is recorded and what is deliberately excluded, why the mechanism binds to the model rather than to one agent harness, and where the approach does not reach.

2. Why Retrospective Documentation Fails

Software engineering has known for decades that documentation written after the fact diverges from the artifact it describes. A classic analysis of design processes concluded that no real project follows the rational sequence its documentation presents, and that such documentation is best understood as a reconstruction produced afterwards [4]. Literate programming proposed the opposite discipline, writing the program as an explanation of itself, and demonstrated that the results can be excellent, but the approach demands sustained authorial effort at exactly the moments when delivery pressure is highest, which limited its adoption [5]. The laws of software evolution supply a structural reason. A system in real use must change continually, so any static description of it begins to decay on the day it is written [6].

The empirical record matches the theory. Surveyed practitioners report that documentation is consulted when it is current and that it rarely is [2]. Field studies find that developers treat the code and their colleagues as the authoritative sources and use documents as a last resort [1].

Decision records [3] improved on heavyweight documentation by shrinking the unit of writing to a single decision and colocating it with the code. What they did not change is the trigger. The record is written when a person remembers to write it, typically after the work, when the alternatives that were rejected and the constraints that forced the choice are already fading. The information that decays fastest, the why and the paths not taken, is precisely the information a later reader needs most. Any capture mechanism that relies on retrospective human initiative inherits this decay.

3. Instrumenting the Write Boundary

kgai's unit of record is a decision, an immutable event pairing its provenance, the author, the date, and the rationale, with the structural mutations it applies to a small graph of domain elements such as features, services, and business objects. Decisions append to a content-addressed log holding one shard per writing installation, and that log is the source of truth. The queryable graph is a deterministic projection of the log, discarded and re-derived at will, following the event sourcing pattern [7]. An element's identity derives from hashing its kind and name in normalized form, so when two writers record the same element independently their records land on one node with no coordination. Nothing is overwritten. A decision

that replaces an earlier one supersedes it, and what was superseded stays in history. The event model and the projection are specified by companion reports in this series (TR-2026-01 and TR-2026-03).

Capture at the write boundary rests on three instruments inside the coding session.

First, a session-start hook injects the effective capture rules into the model's context before any work begins. The rules state what counts as a decision in this project and how elements are named, so a team's conventions arrive with the session rather than living in someone's head.

Second, a skill, a packaged set of instructions the model loads, directs the model to record a structural decision as part of completing any task in which one was made. Recording is framed as part of finishing the work rather than as optional bookkeeping. The cost of articulating the decision is therefore paid at the one moment when it is lowest, while the rationale, the rejected alternatives, and the affected elements are all still in context.

Third, a stop hook runs as the session turn ends. If the model edited code but recorded nothing, the hook prompts it to record before finishing. This closes the failure mode that defeats retrospective documentation, forgetting, with a mechanism rather than a habit.

In headless testing across models, the two-layer design held up. Structural refactors were recorded automatically and reliably, the stop hook captured the decision every time the model was prevented from recording on its own, and trivial edits recorded nothing even when nudged.

4. What Qualifies as a Decision

A capture mechanism that records everything produces a log nobody reads. The capture rules therefore define the unit of record as a structural, domain-level choice, and they define it negatively as well as positively.

The following are recorded: splitting, merging, or moving a module or feature. Renaming a domain element, meaning that its canonical name changes. Changing a dependency, an ownership boundary, or how something is exposed. Deprecating or replacing a prior decision. In each case the record is the choice itself and a short statement of why, linked to the elements it shaped.

The following are not recorded: code-level renames of files, functions, or variables, because the domain element they implement is unchanged. Formatting. Routine bug fixes. The distinction between the two kinds of rename is stated uniformly across the skill and the hooks, after early versions of the rules expressed it inconsistently.

Also excluded, as of the 1.3.0 revision of the capture rules: analyses, research findings, cost or status reports, and recommendations nobody has acted on. When an analysis produces an actual choice, the

choice is recorded with a short rationale, not the analysis. Volatile figures such as prices and counts stay out of the immutable log, because they change without any decision being made.

One category that looks like a non-change is deliberately kept: the dead end. A path that was explored and then given up is recorded together with why it failed, so that neither an engineer nor a model walks it again. In the graph such records are provenance rather than structure. They attach to the element's history without competing with its current head decision.

The result of these rules is a graph that stays small and stable, which is what makes it cheap to rebuild, fast to query, and worth reading.

5. A Skill and a CLI Instead of a Harness Integration

Where the capture logic lives determines what it can survive. kgai binds capture to the language model through instructions, not to a particular agent harness through integration code. The skill and the injected rules are text the model reads. The hooks are the only harness-specific component, and they are thin: inject the rules at session start, prompt at stop.

The write path itself is a standalone command-line interface. A decision is recorded by invoking the CLI with a JSON payload describing the decision and its mutations, and every read, whether recall, history, search, or a conflict listing, is likewise a CLI invocation returning JSON. Any tool that can execute a command can therefore record into and query the same graph, from an editor-embedded agent, from a script, or from a plain terminal.

Two properties of this surface support model-driven use. The CLI's help output explains the model itself, elements, decisions, supersession, conflicts, and the read-before, record-after flow, so an agent that encounters the tool without the skill loaded still receives enough to use it correctly. And the ingest surface rejects a payload containing unknown fields with a message that lists the valid fields and shows how elements are attached, so a model that invents a field is corrected on the spot instead of silently recording a decision that no read command can find.

6. Limitations

The approach has boundaries, and naming them belongs in the same report as the mechanism.

The write boundary is instrumented only where the hooks and the skill run. A change made in a plain editor, in a terminal session without the agent, or by a tool that never loads the skill produces no decision event. The graph observes the instrumented loop, not the repository. For work done outside that loop the fallbacks are manual. A developer or an agent can record the decision afterwards through the CLI, and past decisions, whether old decision records, wiki pages, or tribal knowledge, can be imported in batch,

with each decision carrying its real date so that history and point-in-time queries reflect the true timeline rather than the import time.

Capture works by instruction-following rather than enforcement. The judgment of whether a change is structural or trivial is made by the model applying the rules. The headless results reported above are observations of behavior rather than guarantees, and a model that misjudges the boundary will either miss a decision or record noise. The stop hook narrows the failure to sessions in which code was edited and nothing was recorded. A structural choice that produced no edit in the session is not what the stop hook watches for, so its capture rests on the skill alone.

Finally, two properties of the store shape what capture can deliver. A brand-new graph is empty, so the first real value comes from seeding it with what the team already knows, and until then reads return empty results. And because the graph is deliberately branch-agnostic, a decision recorded on a branch that is later abandoned stays in the graph until a superseding decision retracts it.

7. Conclusion

Retrospective documentation fails because it charges its cost at the moment the payer has the least to gain, after the work, when the reasoning is already dissolving. Four decades of literature, from faked rational processes [4] to decaying documents [2] and code-and-colleague knowledge recovery [1], describe the same temporal mismatch. Agentic development is the first setting in which the write boundary is mechanically observable. The rationale for a structural change sits in a model's context at the moment the change lands, and hooks mark the edges of the session that holds it.

The mechanisms described here are small: injected capture rules, a skill, a stop hook, an append-only log projected into a small graph. What they change is when the writing happens. Captured at the boundary, the why and the rejected alternatives cost little to record and are preserved immutably. We do not claim the loop sees everything. Edits outside it are invisible, and the fallback is manual recording or import. Within the loop, the reliance on human memory that decision records never removed is replaced by a mechanism that runs every session.

References

[1] T. D. LaToza, G. Venolia, and R. DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), 492-501.

[2] A. Forward and T. C. Lethbridge. 2002. The Relevance of Software Documentation, Tools and Technologies: A Survey. Proceedings of the 2002 ACM Symposium on Document Engineering (DocEng 2002), 26-33.

[3] M. Nygard. 2011. Documenting Architecture Decisions. Cognitect blog. https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions

[4] D. L. Parnas and P. C. Clements. 1986. A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering SE-12(2), 251-257.

[5] D. E. Knuth. 1984. Literate Programming. The Computer Journal 27(2), 97-111.

[6] M. M. Lehman. 1980. Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE 68(9), 1060-1076.

[7] M. Fowler. 2005. Event Sourcing. martinfowler.com. https://martinfowler.com/eaaDev/EventSourcing.html