

Jolie Abadir - jba2141

Machine Intelligence (PSYC GU4237)

Professor Jerde

July 2026

Automating Context Management in My LLM Development Workflow

Part 1. Analysis of the workflow

1.1 What this is

I use Claude as one of my main collaborators for implementing my software development projects. I use Claude in the browser as well as Claude Code inside of my text editor, VS Code. Before I propose a more automated workflow, I need to establish an accurate picture of how my current one operates, where it succeeds and where it breaks down. This analysis is grounded in my own artifacts rather than my personal memory. For the two projects I will be using as a case study for this analysis (STEAP, a full-stack study web app, and Workshop, a computer-vision-controlled 3D modeling environment), I keep a structured build-process log in Notion. Each entry records one unit of work against a fixed schema: Entry, Type, Status, Area, Description, Solution, and Lesson Learned. The two logs currently hold 133 entries. Because I had Claude Code actively writing to my Notion database as the work happened, the logs are a rich record of the exact workflow I am studying. This is a single-subject case study with a primary-source paper trail.

1.2 How the workflow runs

The workflow is a loop that I orchestrate. It does not run on its own, but it works autonomously in many aspects of the work. I sit between the model and the project state, forcing my own verification before any commits or pushes are ever done. I'm often reading thousands of lines of code within each session, almost babysitting Claude's work and making sure it doesn't make stupid decisions or stray away from our defined goals. One real unit of work is not a prompt but instead a consolidated collection of context with a specific goal and parameters set around it. A session cycles through this process: pick the task; assemble the context it depends on (for example past log entries, architecture notes, current file state, prior decisions, whatever would change the output); then write the prompt, usually in a fresh window so I can control the scope better; run it, check the output, and write the outcome back as a new log entry. Two properties of this loop account for most of the cost. First, I am the retrieval layer. Deciding what context is necessary for each task and getting it in front of the model is entirely manual. The second is that how well Claude does is gated almost entirely by what context is given and how it's given. The final bit: Notion holds everything, but no part of the session retrieves the context automatically. I give Claude the Notion IDs in my database to pull, as well as keywords to search through the architecture docs and any other implementation documentation. This normally works quite well, especially as over time I have developed context-management habits by trial and error and have achieved a three-prompt split: breaking a change that touches a data table, a backend route, and a frontend UI into three separately scoped prompts. Forcing a fresh window for high-collision changes and diagnosing before fixing remain critical to keeping a clean codebase.

1.3 Where it breaks

The logs let me point at specific, repeating failures that are mostly not the model writing bad code. They are context failures: the right information existed inside of the corpus but was simply not in front of Claude at the right time. These sort into three kinds.

Retrieval failures, where the context existed but did not surface. The clearest issue is the node-id string/integer boundary bug, which by my own note recurred four times on one branch. Session-node rows are keyed on an integer node_id, but new pathway code kept reaching for n.id by analogy to other tables. Each recurrence showed up in a different place: for example, the quiz route and the interrogate hub, then twice more in the Learn pathway (STEAP). A second case quantifies the cost: during a multi-day cloud-sync bug I was dealing with a few months ago, the root cause was a FIFO-ordering bug in the outbox queue. It went undiagnosed for hours, and the lesson I wrote reads plainly that search was available the whole session and not using it cost about 4 hours of speculation.

Staleness failures: the context surfaced but was no longer valid. Some entries were later withdrawn or superseded, and they are dangerous precisely because they read in an almost identical way to the valid ones. For example, one Workshop entry was an explicitly withdrawn rotation regression whose own description says that the original diagnosis was wrong; it was reasoned from data logs without a visual confirmation. A STEAP entry documents the freeform-iframe Learn architecture and notes it being superseded by a block-array restructure. Both are correct to keep as history, and both would actively mislead if pulled up as guidance for the current work. Only the Status field and the timestamps resolve the contradiction; a plain similarity match cannot distinguish a still-valid lesson from a superseded one.

Routing failures, where context is tangled across concerns, are the last issue. When a prompt carries context spanning multiple subsystems, Claude will get tangled. The three-prompt split exists specifically to stop this; the fact that I had to hand-develop that discipline is itself the evidence that context scoping, and not just retrieval, is pulling a large amount of weight within the workflow.

1.4 What the analysis points at

All three failures point back to the same place: the boundary between the store (Notion) and the live context (the conversation), which I currently bridge by hand. The model's reliability is rarely the bottleneck; the bottleneck is rather whether the right, still-valid prior context reaches the model at the

moment of the task. That yields three constraints for anything built to help: automate the retrieval, respect the supersession (using the Status field and timestamps and not just similarity is important for that one), and preserve scoping.

Part 2. Prototype: integrating retrieval into the loop

2.1 What was built

I built a working prototype that automates the store-to-context bridge, and then treated how it retrieves as the object of study. The logs are pulled once from Notion and cached locally as structured records where every schema field is preserved, so the later assessment tests retrieval behavior rather than Notion's latency, and the Status and timestamp signals the staleness constraint needs are still present. Writeback stays manual on purpose: I review Claude's output, commit it, and give Claude the specific instructions to write the log entries to the Notion database. This protects the integrity of the corpus the whole method leans on. Over the cached corpus the prototype runs a retrieval step that returns a scoped, ranked bundle of prior entries for a given task, which is handed to Claude alongside the task in a deliberately thin loop (retrieve, act, verify) in the same way that I already do. The point of interest is the retrieval step itself, which I implemented three ways so their behavior can be compared.

2.2 The three retrieval arms

Arm A is the manual baseline: the current human-run process, no automated retrieval, whose failures Part 1 already documented. Arm B is index-plus-lexical-structured retrieval: a keyword index over the Description and Lesson fields plus structured re-ranking on the schema, so it can prefer settled entries, demote withdrawn or superseded ones, and break ties by recency. This is the approach the file-plus-index memory literature suggests is the strong default at a small, structured corpus size, and it is the arm that can actually use the Status field the staleness constraint demands. Arm C is vector retrieval: it embeds every entry and retrieves by semantic similarity, with no built-in sense that one entry ever replaced

another. I included Arm C specifically so I could observe, rather than assume, the hazard the staleness constraint warns about; where it might add value is on queries phrased differently from an entry's original wording, where keyword matching can fail to match.

Part 3. Would this benefit my workflow?

This part is a reasoned assessment rather than a scored benchmark. I ran the arms on cases drawn from my documented Part 1 failures and then read their behavior against what went wrong, asking a very concrete question: would this prototype have caught the specific failures that cost me time?

3.1 The recurrence failures: yes, clearly

The recurrence failures are the strongest case for the prototype because they are failures of retrieval and nothing else. When I fed the thin loop a fresh node-id boundary task, the retriever returned a scoped bundle led by the two earlier node-id entries, each carrying its own lesson, including the one whose lesson proposes the real fix: a single canonical id-coercion helper would have killed all four instances. This is the context whose absence let the bug recur four times; surfaced automatically at the moment of the task, the four debugging sessions may have reduced to one. The same holds for the 4-hour cloud-sync case: the relevant prior entry existed and was retrievable, and an automated pull would have put it in front of me rather than leaving me to my own devices. For this type of failure the answer is unambiguous: it does not depend on the sophistication of the retrieval method. Instead, both automated arms surfaced the right prior entry, and the manual baseline is, by construction, the process that missed it.

3.2 The staleness failures: only if retrieval respects supersession

The staleness failures are where the arms diverge from each other, and where the assessment is the most important. When I posed a query resembling a stale entry, the vector arm surfaced the withdrawn and superseded entries as readily as the valid ones. This is because similarity has no notion that one entry

replaced another: a withdrawn diagnosis that shares vocabulary with the query is just as retrievable as a live lesson. The lexical-plus-structured arm, using the Status field, suppressed the withdrawn rotation entry, where the correct behavior is to surface nothing live at all, and then demoted the superseded ones. So automating the retrieval only helps here if the retriever is built to respect that supersession; a semantic pull without the same nuance would reintroduce the staleness failure it was meant to prevent. The benefit is real, then, but conditional on the design choice the constraint from Part 1 already flagged.

One case complicated this more than I expected and is worth stating because it does qualify the conclusion above: the fourth node-id instance and the superseded Learn-architecture warning live in the same physical log entry. That entry both teaches the recurrence lesson and describes an architecture that was later replaced. Retrieval that demotes it for being superseded also loses the recurrence lesson; retrieval that surfaces it for the lesson also surfaces the stale architecture. No single Status flag serves both goals, because the entry is genuinely two things existing at the same time. This is just a property of how I write entries, not of the retriever, and it suggests that finer-grained supersession, which marks parts of an entry stale rather than the whole thing, would matter a lot more than the choice of the retrieval method. This is the kind of thing that I would not have seen without building the prototype and watching it behave.

3.3 The routing failures: mostly out of reach

The routing failures are the weakest case for the prototype as built. The three-prompt split is about scoping context across the subsystems, but the prototype preserves scoping in a shallow sense: it returns a bounded, ranked bundle rather than dumping the whole corpus into one window. It does not orchestrate multi-prompt decomposition, which is what actually prevents the schema-route-UI collisions. Retrieval then helps routing indirectly at best, by keeping the bundle small, and the real fix for this failure stays out of scope. I'm not claiming that the prototype addresses routing; it doesn't. It addresses retrieval, it touches staleness if built carefully, and then leaves routing to the discipline I already practice by hand.

3.4 Overall assessment

"Would this benefit my workflow" is the question that this whole paper set out to answer. For the recurrence failures it clearly does, and those are the ones with the measured time cost in the logs. For the staleness failures it does help, but only with supersession-aware retrieval, and even then it is imperfect. For the routing failures, it doesn't really help as it is currently built. The overall result is that the prototype addresses the failure class with the largest documented time cost and the clearest retrieval cause, while leaving the others partly unresolved. Consistent with the file-plus-index memory literature at this corpus size, the lexical-plus-structured arm is the better default here because the corpus contradicts itself and only structured signals resolve the contradictions. The semantic arm's wider matching range is real, but it's also a liability on a corpus that contains stale facts.

3.5 Limitations and what a real evaluation would require

This is a single-subject study: one developer with two project logs, run on one model. The transparency it buys, a contemporaneous, structured, 133-entry trail including entries that record the workflow failing, does not buy generalization; the behavior I observed is a property of this corpus and this way of writing entries. Part 3 is a reasoned assessment against my own documented failures, not a formal measurement, and it should not be read as one. In particular, the retrieval cases were constructed and phrased by me with knowledge of which prior entries were the intended targets, so they are useful for characterizing the behavior and reasoning about my own workflow, but are not a blind test of the retrieval quality. To do a proper evaluation you would need a held-out set of queries written by someone unaware of the corpus, and this would measure whether the right entry returns when the query shares little wording with it. The supersession contrast is the more robust of the observations, since every arm sees an identical query and the difference between them does not depend on my phrasing. Turning this prototype into a measured claim, and pushing scoping from a bounded bundle toward real multi-prompt orchestration so it can touch

the routing failures too, would be the future work that could make this more than an assessment scoped to a single developer's logs.

References

1. Anthropic. (2025, September 29). Effective context engineering for AI agents. Anthropic Engineering Blog. <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>
2. 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>. Code: <https://github.com/letta-ai/letta>
3. LlamaIndex. (2026, January). Did filesystem tools kill vector search? <https://www.llamaindex.ai/blog/did-filesystem-tools-kill-vector-search>

Statement on AI usage

I used Claude to help me analyze my workflow, talk through ideas, and pull resources to draft the new workflow prototype. I also used Claude to help me with the grammar, structure, and flow of my writing.