Never Lose Your Place

Understanding as a Shared Build Artifact:

A Paradigm for AI-Assisted Development

Adam Carlson

June 2026

Abstract

AI-assisted development has quietly inverted a long-standing assumption: that a developer understands a system before building it, or can reconstruct that understanding by reading the code afterward. When a machine writes code faster than any person can internalize it, both halves of that assumption break. But the gap it opens is not the human's alone — the assistant, too, loses its grasp of a system across context-window limits, conversational compaction, and session boundaries. This paper names the resulting condition, the understanding gap suffered by human and AI in complementary ways, and proposes a paradigm for closing it: treat understanding as a *shared* build artifact, manufactured continuously and automatically as a side effect of the work, into a persistent, plain-text substrate that both the human and the AI read, that keeps the two aligned to a single model of the system, and that is kept honest against the source of truth. We state the paradigm's commitments, present an open-source existence proof (*Vibe Scribe*), mark its deliberate exclusions and honest limitations, sketch how it should aid debugging and iteration, and offer predictions by which the paradigm can be judged. We make no efficacy claims: this is a paradigm and a working demonstration, not a measured result.

1 A place lost

On an ordinary afternoon, a development machine restarted itself in the middle of a working session. The change in progress was delicate and half-finished, and a dozen small decisions behind it lived only in the back of the developer's mind and in the volatile context of an AI assistant. By every normal expectation, that thread was gone.

It was not. A single plain-text file — a running journal the assistant had been keeping as a side effect of the work — recorded what had changed, why, and what remained. The next session read that file and resumed at the exact point of interruption, as if nothing had happened.

Note what was recovered. Not only the developer's place in the work, but the *assistant's* model of it — the architecture, the decisions, the dead ends already ruled out. Both had lost the thread; one plain-text file returned it to both. That small, two-sided recovery is the seed of this paper, because the relief it produced points at a problem most developers now feel but few have named.

2 The phenomenon nobody has named

Call it the *understanding gap*: the distance between the code that exists in a project and the understanding of that code that exists in anyone's head.

The gap has two axes. The first is **continuity** — you understood the system a moment ago, and a boundary severed the thread. The second is **comprehension** — you may never have held the thread at all, because the system was assembled faster than you could form a model of it. Neither is a deficiency of skill: a newcomer meets the gap as a wall, an expert as drag, the accumulating weight of code they directed but never fully absorbed.

And the gap is not the human's alone. A human forgets slowly and selectively — across time, divided attention, staff turnover, or never having grasped the system in the first place. An AI assistant forgets abruptly and completely: the context window fills, the conversation is compacted into a lossy summary, the session ends, and its model of the architecture, of what was tried, of the progress made, is simply gone at the next turn. The two forget in *complementary* ways, and the boundaries that sever the thread divide along the same line. Time, handoff, and inattention are where the human loses the thread; the context window, compaction, and the session boundary are where the AI loses its; a reboot or a tool switch takes both at once. Each is a place someone loses their place.

3 The old paradigm, and why it is breaking

Two assumptions sit, unstated, beneath conventional practice. The first is that *understanding precedes code*: you grasp a design, then you implement it. The second is that understanding *can be reconstructed from* code: when the grasp fades, you read the source and recover it. Documentation, under this paradigm, is a separate and largely human effort — written after the fact, for other humans, and left to decay.

AI-assisted development breaks both assumptions at once. When a model generates implementation faster than a person can internalize it, "grasp, then build" collapses: the building outruns the grasping. And "read it back" does not scale to recover what was never deliberately recorded — intent, rejected alternatives, the reason a thing is shaped the way it is. Code is an excellent record of *what*; it is silent on *why*. The old paradigm assumed a human pace of authorship, and a single mind to hold the model. AI-assisted work violates both: the model of the system is now split across two collaborators, neither of whom retains it durably.

4 The new paradigm: understanding as a shared build artifact

We propose a different arrangement. In AI-assisted development, understanding should be **manufactured continuously, as a shared build artifact** — produced alongside the code, by the same machine that writes the code, and deposited into a persistent substrate that both the human and the AI read, kept true automatically as the work proceeds.

Two things follow from the word *shared*. The first is the inversion: understanding stops being a *prerequisite* the developer is assumed to bring, and stops being an *archaeology project* to be undertaken later. It becomes an *output* — a first-class artifact of the build, accruing continuously and available on demand.

The second is less obvious and more powerful. Because both parties read the *same* artifact, they are aligned to the *same* model of the system. This is more than two separately-informed parties; it is a single source of truth for a shared mental model. It converts shared understanding from something renegotiated every session — you explain, the assistant confirms, you both proceed, and tomorrow you do it again — into something durable and co-owned: established once and maintained

continuously. The relationship shifts from *a user instructing a tool* to *two collaborators reasoning from a common, living model.*

"Never lose your place" is the felt promise of this arrangement — and it is a promise to both parties at once. One honest qualifier: the artifact is a shared external *reference* that keeps the two aligned; it is not identical internal cognition. The record holds the common ground; a briefing is where the two reconcile against it — and the paradigm can make that reconciliation *active*, prompting it at the moments the system has just changed, so alignment is restored exactly when it is most at risk of slipping.

5 What the paradigm commits you to

A paradigm is more than a slogan; it implies constraints. Several follow directly from the claim above, independent of any particular implementation.

- **A shared substrate.** The record is for the human *and* the machine — plain text in the project's own repository, readable by a person, parseable by any tool, free of lock-in. Because it is one substrate read by both, it doubles as common ground: the two parties reason from the same names, the same map, the same recorded decisions.
- **Two shapes of memory.** Continuity and comprehension want opposite structures. A *map* — revised in place — describes the system as it is now; this is what lets you grasp it. A *log* — append-only, never edited — preserves the temporal thread: not every edit, but the *decisions*, the *reasons* behind them, the *alternatives rejected*, and the *surprises*. It records why the system came to be the way it is — the one thing the code, a record only of its latest state, can never recover — and it is written for a reader who was not present, pre-eminently the next AI session. Its most valuable service is to keep that reader from *repeating itself*: re-exploring a dead end, re-proposing a rejected approach, re-introducing a fixed bug. Same maintenance habit, opposite update semantics; conflating them destroys both.
- **Maintenance as a side effect.** If keeping the record current is a separate task the human must remember, it will not happen. The substrate must be maintained automatically, as a byproduct of the work itself.
- **Honesty about staleness.** A record that cannot distinguish what it has verified from what may have drifted is worse than none, because it invites false confidence. Freshness must be a first-class signal the record carries and surfaces.
- **Self-verification.** The understanding must be checkable against the source of truth. The machine proposes the record; the code verifies it — the record audits itself against the code rather than decaying unchecked.
- **A shared vocabulary.** A person cannot navigate a system they have no names for. The substrate coins memorable, human-readable names for the system's parts, anchored to real code, so a single handle indexes across the map, the history, and the source — and means the same thing to the human and the AI.

6 An existence proof: *Vibe Scribe*

Vibe Scribe is one realization of the paradigm, open-source and runnable today.[1] It is not the paradigm; it is evidence the paradigm is buildable rather than aspirational.

It keeps two plain-markdown documents in the repository: a *dev journal* (the log) and a *system overview* (the map, which carries a *Lexicon* of coined names anchored to the code). The documents are maintained automatically: a lightweight, fail-open hook notices when source code changed but the documents did not, and quietly asks the assistant to record the change before finishing. The map carries per-entry verification dates, so stale sections announce themselves; an audit command re-checks the map against the code and refreshes or flags it. A briefing command reads the documents back in plain language — with small text diagrams — so the project can explain itself to its author, and so the assistant can re-ground itself at the start of a session and reconcile with the user on the current state. After a significant change, the assistant can go further and *offer* such a briefing unprompted, framed as an intent check — "does this match what you had in mind?" — surfacing any gap between what was intended and what was built while it is fresh and cheap to correct. The same core runs across three AI coding tools (Claude Code, OpenAI Codex, and Cursor) from one shared implementation, because the artifact it produces is tool-agnostic text. Each of these is a tenet from the previous section, made concrete; none requires research-grade machinery, and the result belongs to the project, not to any vendor.

7 What the paradigm excludes

A lens is defined as much by what it leaves out. *Vibe Scribe* declined a formal knowledge graph of the system's components: the relationships it would encode are already captured, in human-readable form, by the map and its vocabulary, and a machine-only graph is a lossy second copy of something the code already encodes precisely and that static analysis can extract for free. A graph earns its keep when the graph *is* the product (as in a graph of ideas with no other representation), not when it shadows source that already exists. For the same reason it declined a rigid relationship ontology, preferring open, plain-language relationship labels, and declined rendered or binary diagrams in favor of plain text that diffs, travels, and reads anywhere. The paradigm favors the representation a human can read and a tool can verify over the one that is merely machine-precise.

8 Honest positioning

This paradigm stands on familiar ground, and says so — but the prior art divides almost exactly along the line this paper has drawn. Engineering notebooks, architecture decision records, runbooks, README-driven development, living documentation, and domain-driven design's notion of a shared ubiquitous language all serve *human* understanding. The "memory" files, context-management schemes, and retrieval-over-conversation emerging around coding agents serve *machine* understanding. Each side addresses one party's forgetting and ignores the other's.

What is new here is not a component but a *unification*: a single substrate that serves both, maintained by the AI, read by both, and — the decisive part — that keeps the two *aligned* to the same model rather than letting their understandings drift apart. The recurring failure of the prior art is not conception but sustainability: the records rot because maintenance is manual and

[1] Source and documentation: `https://github.com/adamccarlson/vibe-scribe` (MIT-licensed).

staleness is invisible. The contribution is to move that discipline into the build loop, to make the record honest about its own freshness, and to make it the common ground of a human–AI pair.

We claim no measured benefit. *Vibe Scribe* is new; it has not been evaluated; it leans in part on the assistant following instructions; verification is harder to sustain as a system grows; and its automatic enforcement has not been exercised live in every supported tool. This paper presents a paradigm and a working demonstration of it — not evidence that it improves any outcome. That honesty is itself a commitment of the paradigm: a record, including this one, should not claim more confidence than it has earned.

9 What it should make easier: debugging and iteration

A paradigm should pay rent in practice, and this one makes specific — though, we stress, as-yet-unmeasured — promises about the two activities that consume most of a developer's time: fixing what is broken and improving what works. At least five mechanisms are distinct.

- **Localization.** Most defects arrive with recent changes, and the log is precisely a record of recent change and its rationale — the first question in debugging, answered directly.
- **Blast-radius reasoning.** The map's relationships — which component owns which data, what calls what — bound where a symptom can originate and reveal what a fix might break.
- **Compounding failure memory.** A diagnosed fault, once recorded, is thereafter looked up rather than re-diagnosed, so debugging knowledge accumulates instead of evaporating.
- **Non-repetition.** Most pointed for the AI: because the record holds the *why* and the rejected paths, neither party "fixes" a problem by reverting to an approach already known to fail, or by violating an invariant an earlier decision was protecting.
- **Architecture-respecting change.** An accurate, shared model lets improvements fit the existing design rather than fight it, and lets human and AI reason about the change from the *same* model in the *same* vocabulary — fewer mismatches of the form "I thought that component did X."

These are affordances the paradigm should provide, not results it has demonstrated. Whether they hold is what the experience report below would test.

10 What the paradigm predicts

A paradigm worth the name makes claims that could turn out false. If understanding-as-a-shared-build-artifact is the right frame, several things should follow. The binding constraint in AI-assisted development should shift from *writing* code to *comprehending and handing off* code. Tooling that manufactures understanding should become as ordinary as version control. AI coding agents should come to depend on durable external substrates that they themselves maintain — and the artifact that serves the human's comprehension and the one that serves the agent's continuity should converge into a single shared record, rather than living in two separate tools. And teams that skip all this should accumulate a compounding "comprehension debt": shippable systems that nobody, human or machine, can safely change.

The paradigm is also disconfirmable. If models become capable enough that code never needs to be

understood by a person again, or if static analysis can reconstruct intent — not just structure — fully and on demand, then manufacturing understanding is wasted effort and the paradigm is moot. The honest next step is neither argument nor assertion but use: run the approach on real projects over real time, and report what actually happens. That experience report, not this paper, is where evidence begins.

11 The smallest promise, the largest stakes

A machine rebooted, and neither the developer nor the assistant lost their place. That is the smallest possible version of the claim, and it is also the whole of it. As AI writes more of the code, understanding stops arriving for free, and it stops being something either mind reliably holds alone. It has to be manufactured — continuously, automatically, honestly, and *together* — or it simply will not be there when the thread breaks, as it always eventually does.

Here is one paradigm for manufacturing it, and one proof that it can be built. Never lose your place — for either of you.