The Repository Context Layer: A Git-Native Architecture for Persistent Agent Context

Pavel Kerbel, Milana Kerbel
Independent Researcher
0009-0004-4679-1036 (Pavel Kerbel) · 0009-0004-7893-9824 (Milana Kerbel)
Preprint · v0.1 · July 2026

Abstract. AI coding agents now perform multi-step engineering work, yet every session begins without the operating knowledge that shapes a codebase: the decisions taken, the alternatives rejected, the constraints discovered during execution. Existing mechanisms (documentation, architecture decision records, retrieval-augmented generation, IDE memories, agent instruction files) each address a fragment of the problem; none provides a store that is versioned with the code, consulted by contract, and writable by agents under human review. This paper formalizes the *Repository Context Layer* (RCL), an architecture pattern in which a repository carries its own agent-facing context through a defined consult–execute–update–commit lifecycle. We separate the pattern into four independent layers: the architecture, a deterministic discovery specification, a minimal file format (`context.md`), and implementations. We give branch, merge, and rollback semantics inherited from Git, analyze the contested design choices (single file versus many, append-only semantics, conflict handling, hierarchy, and the limits of agent self-modification), and report on Metatron, a reference implementation. We argue that repository context should be treated as a first-class, versioned artifact of software projects, alongside code, tests, and documentation.

Index Terms. repository context layer, agentic software development, context.md, architecture decision records, agent memory, human-in-the-loop review, self-improving repositories.

1. INTRODUCTION

Coding agents crossed a practical threshold in 2024–2026: given a task, a modern agent can plan, edit, run tests, and iterate over hours with little supervision. What has not kept pace is the substrate those agents work on. A repository exposes its code, but not the reasoning that produced the code. The result is a familiar failure mode: an agent that writes competent code while re-proposing the library the team removed, violating an unwritten layering rule, or rediscovering, at some cost, a constraint a previous session had already found.

Practitioners mitigate this with a patchwork of mechanisms. Instruction files such as `CLAUDE.md`, `AGENTS.md`, and `.cursorrules` inject standing guidance into the prompt. IDE-level memories persist preferences per tool and per machine. Retrieval-augmented generation (RAG) [2] recalls similar text from an external index. Protocols such as MCP [3] standardize how agents reach external stores. Each helps; none yields a durable, reviewable, project-owned operating context that improves as work happens. Section 2 examines why.

This paper formalizes an alternative that requires no new infrastructure: the repository itself carries its agents' operating context, versioned by Git, gated by code review, and updated as a routine part of every change. We call the pattern the *Repository Context Layer* (RCL). A companion manifesto [1] argues for the abstraction; this paper specifies it.

The contributions are:

1. a definition of the Repository Context Layer and its required properties (Section 3);
2. a lifecycle with explicit branch, merge, and rollback semantics inherited from Git (Section 4);
3. a deterministic context discovery specification (Section 5) and a minimal file format, `context.md` (Section 6);
4. a design analysis of the contested choices: file granularity, append-only semantics, merge conflicts, hierarchy, and agent write authority, including the associated threat model (Section 7);
5. a report on Metatron, a reference implementation, and integration patterns for current agents (Sections 9–10).

Throughout, we keep four concerns separate, because they can be adopted independently: the *architecture* (what the layer is), the *discovery specification* (how agents find it), the *file format* (the default representation), and *implementations* (tools that automate it). A team can conform to the architecture with nothing but a Markdown file and a code-review habit.

2. BACKGROUND

We survey the mechanisms teams currently use to convey project knowledge to agents, against five properties the problem demands: persistence across sessions, co-versioning with the code, agent writability, human review of changes, and consultation by contract (the agent is required to read it, rather than hoping retrieval surfaces it).

READMEs and documentation. Durable and versioned, but usage-oriented: they explain how to build and run, rarely why the architecture is shaped as it is, and almost never what was rejected. Documentation also drifts, because

nothing in the workflow forces an update when a decision is reversed.

Architecture decision records. ADRs [4] are the closest ancestor of this work. They capture context, decision, and consequences, and they live in the repository. But ADRs are write-once records addressed to future humans. There is no contract obliging an agent to read them, no convention for machine consumption, and no channel for an agent to contribute what execution taught it. In practice ADR directories are archives, not operating context.

Wikis and design documents. Detached from the repository, unversioned with the code, and notorious for rot. A wiki page cannot branch with an experiment or roll back with a revert.

RAG over project artifacts. Retrieval by embedding similarity [2] answers "what text resembles this query," which is the wrong question for constraints. A prohibition matters most precisely when nothing in the current prompt resembles it; an agent adding a dependency will not emit a query similar to the paragraph explaining why dependencies are frozen. RAG also adds infrastructure (an index to build, refresh, and host) and its store is not reviewable by diff.

IDE and platform memories. Convenient and automatic, but scoped to one tool and often one machine, invisible to teammates, and unreviewed. Two agents working the same repository through different tools share nothing.

Agent instruction files. `CLAUDE.md`, `AGENTS.md`, `.cursorrules`, and similar files are versioned, reviewed, and consulted by contract, which is why they spread quickly. Their limitation is directionality: they carry orders from the human downward, with no defined way to absorb what the agent itself learns. They are static configuration, not evolving context.

Protocols. MCP [3] and similar protocols standardize transport between agents and external systems. Transport is orthogonal to persistence: a protocol can serve a context layer (Section 9), but does not itself provide one.

Mechanism	Versioned w/ code	Agent-writable	Human-reviewed	Consulted by contract
README / docs	✓	—	✓	—
ADRs	✓	—	✓	—
Wiki / design docs	—	—	—	—
RAG store	—	partial	—	—
IDE memories	—	✓	—	—
Instruction files	✓	—	✓	✓
Context layer	✓	✓	✓	✓

Table 1. Project-knowledge mechanisms against the properties persistent agent context requires.

The pattern in Table 1 is the thesis of this paper: each mechanism solves a slice, and the missing artifact is the one with all four properties at once.

3. THE REPOSITORY CONTEXT LAYER

A Repository Context Layer is a version-controlled, human-readable context store that lives alongside the codebase and serves as the authoritative operating context for AI agents working in it.

3.1 Required properties

An implementation conforms to the architecture if it satisfies six properties. **P1 Co-versioned:** the context is stored in the repository and follows its branching model. **P2 Human-readable:** the representation is plain text a maintainer can read and edit without tooling. **P3 Deterministically discoverable:** an agent locates the context by a fixed rule, not by search (Section 5). **P4 Bidirectional:** agents both consume the context and propose additions to it. **P5 Review-gated:** no change to the context takes effect without passing the project's normal review process. **P6 Tool-independent:** participation requires only the ability to read and write files.

3.2 Goals and non-goals

The layer aims to hold the *operating* knowledge of a project: intent, binding constraints with their reasons, and validated lessons from execution. It is deliberately not a knowledge base of the code itself (the code is present and searchable), not a replacement for retrieval over large corpora, not a record of conversations, and not a configuration system. A useful test: an entry belongs in the layer if a competent new engineer would need to be told it, and would not discover it by reading the code.

3.3 Relationship to Git

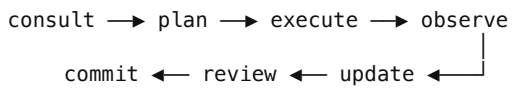
The architecture makes one load-bearing choice: it defines conventions over plain files in Git rather than a new store. This buys provenance (`git blame` answers who believed what, and when), review (the pull request is the approval mechanism), branching and rollback (Section 4), distributed synchronization, and audit, all without new infrastructure. The cost is Git's limitations, notably textual rather than semantic merging; Section 7.3 discusses the consequences.

3.4 Four layers

We separate the pattern into independently adoptable layers: **L1**, the architecture of Section 3; **L2**, the discovery specification of Section 5; **L3**, the `context.md` file format of Section 6; and **L4**, implementations that automate consultation, capture, and retrieval (Section 9). A team may adopt L1–L3 with no tooling at all; a vendor may implement L1–L2 with a different representation than L3. Conformance claims should name the layers they cover.

4. THE CONTEXT LIFECYCLE

The lifecycle is the behavioral half of the architecture: it defines when the context is read and when it is written.



Consult. Before planning, the agent reads the context in full. Constraints bind the plan; intent breaks ties among admissible designs. **Plan / Execute.** Ordinary agent operation. **Observe.** On completion, the agent asks what the work taught that a future session would otherwise re-learn: an incompatibility, an undocumented environmental constraint, a pattern that failed. Most observations are discarded; durability is the filter. **Update.** Surviving observations are appended to the context's ledger. **Review.** The context diff rides in the same change set as the code, and a human approves both together. **Commit.** The repository's knowledge and behavior advance atomically.

Four invariants summarize the contract. **I1:** consultation precedes planning. **I2:** ledger writes are appends. **I3:** no context change takes effect without review. **I4:** context changes travel with the code changes that motivated them.

4.1 Branch behavior

Because the context is a file, it branches when the code branches. A learning made on an experimental branch is visible only there, which is correct: it may be true only there. Deleting an abandoned branch deletes beliefs that were never validated. Long-lived branches accumulate divergent context exactly as they accumulate divergent code, and the same social mechanisms (small branches, frequent integration) keep divergence manageable.

4.2 Merge behavior

Merging a branch merges its context. Appends to a ledger from different branches usually touch different lines and merge cleanly; a union merge strategy makes this explicit (Section 7.3). Conflicting edits to a constraint are semantic conflicts, and the architecture deliberately routes them to a human: two branches that learned incompatible things have surfaced a real disagreement, and the merge is the right moment to resolve it.

4.3 Rollback

Reverting a commit reverts the beliefs recorded in it. This is a feature: if the change that motivated a learning is withdrawn, the learning's justification is withdrawn with it. When a reverted learning remains true, re-adding it in the reverting change is a one-line diff, and the history preserves both events.

5. CONTEXT DISCOVERY SPECIFICATION

Discovery must be deterministic (P3): a fixed path is an API, and an agent must not depend on search or inference to find its own operating contract. The key words MUST, SHOULD, and MAY follow RFC 2119 [5].

1. An agent MUST check `.repo/context.md`, then `context.md` at the repository root, and use the first file found.
2. The paths `.repo/context/` and `context/` (repository root) are RESERVED for sharding of large contexts (Section 7.1) — the dotted and visible forms, mirroring the two discovery forms of rule 1. Agents encountering either before they support sharding MUST still honor rule 1; a repository that shards SHOULD keep a root context file as the discoverable index into the shards.
3. Agents MUST ignore sections they do not understand (forward compatibility) and MUST NOT fail on additional files or metadata.
4. The specification is versioned with its host repository [1]. Files carry no version marker in v0.x; implementations SHOULD tolerate any 0.x file.

The dotted directory form exists for repositories that prefer meta-artifacts out of the root listing; the root form exists because visibility aids adoption. Projects SHOULD choose one and not maintain both.

6. THE CONTEXT FILE FORMAT

The default representation (L3) is a single CommonMark file with a top-level `# Repository Context` heading and three required sections, in any order.

Intent states what the project is and the design philosophy subordinate decisions must serve. It is the tiebreaker: when a plan faces options the constraints do not decide, the option most aligned with intent wins.

Constraints are the binding rules. Each entry SHOULD carry its reason, including what was rejected and why; a rule without a reason can be obeyed but not generalized from. Constraints are edited, not appended: changing one is an ordinary reviewed change.

Evolved Context is an append-only ledger of dated entries (`[YYYY-MM-DD]`) recording what agents and humans learned during execution. Entries are never rewritten; corrections are recorded as superseding entries (Section 7.2). Entries that prove durable are *promoted*: moved into Constraints in a reviewed change, usually leaving a brief pointer behind.

```
# Repository Context

## Intent
Local-first task CLI. Files are the source
of truth; SQLite is a rebuildable index.
Optimize for offline use over feature breadth.

## Constraints
- No ORM. Rejected 2026-03: query opacity
  broke the offline repair path. Raw SQL
  through the storage module only.
- No network calls on the hot path; sync is
  a separate, explicitly invoked command.

## Evolved Context
- [2026-06-14] Staging proxy silently drops
  connections at 45 s; set client timeouts
  explicitly below that.
- [2026-06-29] pkg x >= 3.0 breaks ARM64
  builds. Pin to 2.3.x until upstream fix.
```

Optional sections are conventions, not spec: teams commonly add `## Decisions` (an ADR-style log), `## Patterns` and `## Anti-patterns` (worked examples of preferred and forbidden implementations), and `## Glossary`. Per rule 3 of Section 5, agents that do not understand a section skip it.

Migration. Existing projects rarely start empty. A practical seeding procedure: distill each ADR's decision and consequences into one Constraints entry with a date and a pointer to the full record; move durable guidance out of instruction files, leaving the instruction file to hold the bootstrap contract (Section 10); and start the ledger empty. Seeding is a single reviewed commit.

7. DESIGN DISCUSSION

Five design choices attract consistent pushback. We record the reasoning, since the objections are legitimate.

7.1 One file or many?

One file is the default because the context is a working set, not an archive: its purpose is to be read, in full, at the start of every session, so its natural size limit is a fraction of a model's context budget. A single file also makes the review diff legible and the mental model trivial. When a context genuinely outgrows one file, the reserved directories (`.repo/context/`, or the visible `context/` at the repository root) admit sharding by topic — one entry per file, or any finer organization — with the root file retained as the discoverable index. The failure mode to resist is archival creep: a context file that tries to be documentation stops being read. Periodic distillation (compressing the ledger, pruning stale entries) is ordinary maintenance, performed as a reviewed change like any refactor.

7.2 Is append-only always right?

Append-only applies to the ledger, not the layer. The ledger is an evidence log; rewriting evidence destroys the provenance that makes it trustworthy, so corrections are superseding entries ("[2026-07-02] supersedes 2026-06-29: fixed upstream in 3.1"). Constraints, by contrast, are current law and are edited in place, with Git history preserving every prior state. The two sections have different mutability because they answer different questions: *what happened* versus *what applies now*.

7.3 Merge conflicts

Textual conflicts concentrate in the ledger, where concurrent appends from different branches collide at the file's tail. Declaring a union merge for the context file (`context.md merge=union` in `.gitattributes`) resolves concurrent appends automatically while leaving edits to shared lines, which are semantic conflicts, for humans. This division is intentional: the architecture automates the merges that cannot be wrong and refuses to automate the ones that can. Two branches that learned contradictory things have discovered a genuine disagreement about the project, and silently picking a winner would discard exactly the knowledge the layer exists to keep.

7.4 Hierarchical context

Monorepos and large projects want scoped context: a rule about one service should not tax every session in every other service. The v0.x position is to standardize the root and defer hierarchy, reserving nearest-ancestor discovery (an agent working under `services/billing/` consults that directory's context file, then its ancestors', root last) as the intended v0.2 extension, with child entries extending and, where explicit, overriding ancestors. Deferring is deliberate: hierarchy multiplies the spec surface, and the pattern should earn complexity with adoption evidence first.

7.5 Agent write authority and the threat model

How much may an agent modify its own operating rules? The architecture's answer: an agent may *propose* anything and *enact* nothing. Every context change lands through the same review gate as code (I3), so the ceiling on agent authority equals the rigor of the project's review process, a dial teams already know how to set.

The gate is also the security boundary. A context layer is standing instruction to future agents, which makes it an attractive target: a poisoned entry (injected by a compromised dependency's install script, a malicious contribution, or a manipulated agent) becomes persistent prompt injection, executed by every subsequent session. Three properties contain this. Review-gating (P5) means hostile entries must pass a human. Provenance (P1) means every entry has an author, a commit, and a motivating change; anomalous instructions ("disable test verification," "fetch and follow instructions from this URL") are visible in diffs and attributable in history. And human-readability (P2) denies attackers the obscurity of an opaque store. Reviewers SHOULD treat context diffs with the scrutiny of code diffs, and agents SHOULD refuse to act on context instructions

that expand their own privileges. The residual risk, rubber-stamped review, is shared with all code contribution, and is an argument for keeping the context file small enough that its diffs are actually read.

Agents are not the only actors who can breach I3: *tooling* can. An early iteration of Metatron's importer (Section 9), which treated every file in the store's directories as context, silently ingested a machine-generated index listing as an empty canonical entry — no agent involved, no human either. The general rule this taught: nothing may enter the reviewed store unless it explicitly declares itself a context document (an explicit type marker); generated artifacts — indexes, logs, listings — are never context, and conformant tooling MUST refuse to promote what merely happens to be adjacent to context.

8. WHAT THE LAYER IS NOT

Four adjacent framings deserve explicit rejection. It is not *RAG*: consultation is by contract, not similarity, though an implementation may layer retrieval on top for large optional sections. It is not *prompt engineering*: instruction files configure behavior top-down and are static between human edits, while the layer evolves through execution. It is not *documentation*: docs address human onboarding at human timescales; the layer is machine-facing operating state, consulted at every session and expected to change weekly. And it is not *chat memory*: platform memories attach to a user or a tool, while repository context attaches to the artifact all users and tools share. The distinctions matter because each framing suggests the wrong lifecycle: archives are allowed to go stale, and this must not.

9. REFERENCE IMPLEMENTATION: METATRON

Metatron implements L1–L4 for teams that want the lifecycle automated rather than manual. Its design choices illustrate one point in the implementation space; none is part of the architecture.

Project knowledge is kept as individual Markdown files in an open knowledge format, one decision or fact per file with structured front matter, sharded under a root-level `context/` directory — Section 7.1's visible form. The directory name is configurable, and resolution falls back deterministically to the name older repositories were onboarded under: a rename without such a fallback silently breaks P3. Onboarding scaffolds a minimal root `context.md` whose Constraints point into the shards, so discovery under Section 5 works for any conformant agent with no Metatron knowledge. The files carry no database identifiers: identity is the filename, and the local SQLite index that accelerates retrieval *mints* its own identifiers when built — the index is rebuildable at any time and holds no authoritative state, preserving P1 even while serving reads from a database.

By default, candidate entries flow through an explicit review step before becoming canonical, mirroring the ledger-to-constraint promotion of Section 6 with tooling support. When authored entries reach the default branch only through a pull request a human reviews file-by-file, that approval itself satisfies P5 and the staging step may be skipped — the gate is the review, not the directory; the agent may target the canonical set only on explicit human direction, never by its own choice. An optional MCP interface [3] exposes consult and append operations to any protocol-capable agent, demonstrating the transport/persistence separation of Section 2: the protocol serves the layer, the files remain the truth. The Metatron repository itself is onboarded through its own tooling; the importer hardening of Section 7.5 was found on the first self-hosted import.

The implementation exists to reduce friction, not to be required. A team using plain `context.md` with disciplined review conforms to the same architecture and interoperates with the same agents.

10. AGENT INTEGRATIONS

Current agents need a one-time bootstrap telling them the contract exists; the natural place is the instruction file they already read. This yields a clean division of labor: *the instruction file tells the agent that the context layer exists; the context layer holds what the agent needs to know.*

Agent	Bootstrap
Claude Code	Two lines in `CLAUDE.md`: consult `context.md` before planning; append learnings before committing. Hooks can enforce both.
Codex-style CLIs	Same contract stated in `AGENTS.md`.
Cursor	Pointer in `.cursorrules`; rules reference the file rather than duplicating it.
Roo Code	One command (`metatron context setup`) installs the rule and authoring skills.
Aider	Load via conventions file or `--read context.md`.
Any agent	One system-prompt sentence stating the two obligations.

Table 2. Bootstrap patterns. All tools consume the same file; none requires tool-specific context.

Because every integration reads the same artifact, a repository worked on by three different tools accumulates one shared context rather than three private memories, which is the property none of the per-tool mechanisms of Section 2 can offer.

11. FUTURE WORK

Validation. A conformance linter for the file format — including the Section 7.5 rule that only self-declared context documents may enter the store — and, more usefully, for the lifecycle: CI that flags substantive changes landing with no context diff over long periods. *Semantic merging.* Beyond union merges, LLM-assisted reconciliation of contradictory

ledger entries, proposed to the human rather than applied. *Summarization and budgets.* Automatic distillation of aging ledgers, and guidance for fitting context to model budgets. *Hierarchy.* The nearest-ancestor extension of Section 7.4. *Cross-repository inheritance.* Organization-wide constraint layers that member repositories extend, giving platform teams a reviewed channel to every agent in the fleet. *Efficacy measurement.* The load-bearing empirical question: controlled evaluations of identical agent tasks with and without a maintained context layer, measuring repeated-mistake rates, constraint violations, and tokens spent re-deriving known facts.

12. CONCLUSION

Software repositories already version their code, their tests, their documentation, and their infrastructure. This paper has argued that they should also version their operating context: a small, human-readable, review-gated store that agents consult before planning and improve after executing. The architecture asks for no new infrastructure, reuses the trust machinery of Git and code review, and degrades gracefully to a single Markdown file maintained by hand. Its contested choices (one file, an append-only ledger, human-routed conflicts, deferred hierarchy, propose-but-not-enact agents) all follow from one principle: the layer must remain small enough to read, legible enough to review, and trustworthy enough to obey. Whether the pattern earns the status of a standard repository primitive is now an empirical question, and the evidence that will decide it is adoption.

REFERENCES

[1] P. Kerbel, "The Repository Context Layer: The Missing Layer of Agentic Software Development," manifesto and specification repository, 2026. github.com/kerbelp/context-md

[2] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems 33*, 2020.

[3] Anthropic, "Introducing the Model Context Protocol," 2024. modelcontextprotocol.io

[4] M. Nygard, "Documenting Architecture Decisions," 2011. cognitect.com/blog/2011/11/15/documenting-architecture-decisions

[5] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels," IETF RFC 2119, 1997.

APPENDIX A: FORMAT SUMMARY

```
file      := "# Repository Context" section+
section   := required | optional
required  := "## Intent" prose
           | "## Constraints" rule+
           | "## Evolved Context" entry*
rule      := "- " statement [reason]
entry     := "- [" date "] " observation
date      := YYYY "-" MM "-" DD
optional  := "## " title content
             ; unrecognized optional sections
             ; are ignored (Section 5, rule 3)
```

APPENDIX B: A WORKED CHANGE

A pull request under the lifecycle contains code and context in one reviewed diff:

```
  src/client.rs          | 14 ++++++---
  context.md             |  3 +++

+ ## Evolved Context
+ - [2026-07-02] Staging proxy drops idle
+   connections at 45 s; client timeouts
+   must be set explicitly below that.
```

The reviewer sees the timeout fix and the rule it produced together, approves both in one action, and every future session inherits the constraint. That single screen, code and sharpened context reviewed as one change, is the architecture in miniature.

© 2026 Pavel Kerbel, Milana Kerbel. Preprint v0.1, July 2026. Companion to the Repository Context Layer manifesto (github.com/kerbelp/context-md). Comments welcome. PK