Workflow Memory for Long-Horizon Agentic Composition: Architecture, Dual-Mode Retrieval, and Retrieval-Correction Signal

Vladimir Ovcharov*
LEX AI LLC, Kyiv, Ukraine
https://legal.org.ua

May 2026

Abstract

Sixty percent of context tokens in current LLM agentic sessions are wasted—redundant re-explanation of decisions already made in prior sessions. Baseline measurements from 304 production sessions confirm a median bootstrap cost of 30,115 input tokens with a 60% context waste ratio, while existing memory systems [6, 18, 19] organize around dialogue episodes and code-RAG [21, 24] retrieves without decision provenance, leaving this waste unaddressed.

We present a *three-layer workflow memory* architecture—domain, workflow, and practitioner layers with distinct retrieval semantics—targeting ≤10K bootstrap tokens and ≤20% waste. The key architectural primitive is *dual-mode retrieval*: pull mode queries all layers at session start; push mode refreshes memory for dormant tasks proportional to repository activity, so that context remains current even after weeks of inactivity. The third contribution reframes the memory layer as scalable oversight infrastructure: *retrieval-correction edits*—practitioner corrections that would have been unnecessary had memory surfaced relevant context—constitute a process-level oversight signal that is denser than outcome-level supervision and scales with agent autonomy [17].

The architecture is deployed on a production legal-technology platform (70+ MCP tools, 380M+ records, 1,547 merged PRs in 105 days). The memory layer does not merely consume alignment data—it produces it: every retrieval failure that triggers a practitioner correction simultaneously generates a preference pair for the companion alignment pipeline [17].

1 Introduction

A single practitioner, working recursively with Claude Code as the primary agentic engineering counterpart, shipped 1,547 merged pull requests across seven interconnected projects in 105 days [17, §1]. The system under construction—a legal-technology platform with 70+ MCP tools, 380M+ indexed records, and production customers—is not a toy benchmark. It is an operational regime where architectural decisions made in week two constrain implementation choices in week six, where validator definitions evolve alongside the code they protect, and where the practitioner's correction history encodes domain expertise that no context window can hold indefinitely.

This operational regime is not served by existing memory or retrieval approaches.

*Correspondence: volodymyr@legal.org.ua. Repository: https://github.com/overthelex/SecondLayer.

CLAUDE.md scales as hand-curation. The dominant workaround for cross-session context in agentic workflows is a flat Markdown file (`CLAUDE.md`) that accumulates project conventions, architectural notes, and operational constraints. In practice, its size grows linearly with project age: over 85 days and 25 commits, the `CLAUDE.md` for the platform described above grew from 4,099 to 24,148 characters (474 lines) at a rate of 217 characters/day ($R^2 = 0.87$; see Section 7.2). A new session loads the entire file regardless of task relevance, wasting context budget on content that is irrelevant to the current task and out of date with respect to recent changes. Worse, the practitioner bears the curation burden: every architectural decision must be manually distilled into the file, and stale entries silently degrade session quality.

Long context is not the answer. Models with 1M-token context windows [20] appear to eliminate the need for retrieval. Empirically, they do not. Attention degradation past ∼200K tokens is well documented: the "lost in the middle" effect causes models to underweight information in the interior of long contexts [14], and synthetic benchmarks like RULER confirm that effective context utilization degrades with length [7]. Even where the context fits, cost scales with total context length, not with the relevance of individual entries. At the 6-week horizon, the accumulated working set—prior decisions, rejected alternatives, validator evolution, constitutional principle invocations—exceeds what fits efficiently in any context window.

Conversational-agent memory is the wrong shape. MemGPT [18], Generative Agents [19], A-MEM [22], and Mem0 [6] organize memory around dialogue episodes: episodic recall of prior conversations, semantic memory for persistent facts, and reflection for self-assessment. The retrieval unit is the conversational turn or the character observation. None of these systems is designed for decision provenance—the record of which architectural alternative was chosen over which other, which constitutional principle was invoked, and which validator caught which class of regression.

Outcome-level supervision becomes thin at scale. As coding agents take on longer-horizon work—Anthropic Engineering documents agent sessions spanning thousands of context windows on a single project [3]—outcome-level signal (did the shipped code work?) becomes a sparse, lagging indicator. A single binary outcome may correspond to hours of agent activity and dozens of architectural micro-decisions. Anthropic's published research priorities identify scalable oversight as a central open problem: how to design oversight mechanisms whose signal density grows with agent capability rather than degrading [2]. This paper contributes a substrate for one such mechanism. The memory layer's retrieval-correction edits are process-level oversight signals: each one identifies a specific gap between the context the agent had and the context the practitioner had, localized to a single edit. Unlike outcome-level signals, retrieval-correction signals are produced continuously throughout a session, and their generation rate is invariant to session duration.

This paper provides the mechanism for two conditions stated but not mechanized in the companion paper [17, §2]: (1) the codebase as persistent context, and (2) compositional task layering across multi-week horizons. The contributions are:

1. A **three-layer memory decomposition**—domain, workflow, and practitioner—with distinct retrieval semantics per layer (Section 4).
2. **Dual-mode pull/push retrieval** as a first-class architectural primitive, where push-mode refresh keeps dormant-task memory current without requiring active sessions (Section 5).
3. **Retrieval-correction edits** as a process-level oversight signal that scales with agent autonomy (Section 5.4). Each retrieval-correction edit localizes a specific context gap; aggregated across sessions, these edits constitute a dense oversight signal that complements the sparse outcome-level

signal used in standard RLHF, and connects to the alignment pipeline described in the companion paper [17].

The evaluation plan (Section 7) defines measurable targets against instrumented baselines from the companion paper's dataset. The architecture is designed for a single-practitioner regime and does not claim generality to multi-developer teams; limitations are discussed in Section 9.

2 Related Work

2.1 Memory for Conversational Agents

The recent proliferation of LLM agents has produced a family of memory architectures organized around dialogue persistence. MemGPT [18] virtualizes a memory hierarchy analogous to an operating system's paging mechanism: a fixed main-context window is backed by an unbounded "archival storage" that the agent can read from and write to via self-directed function calls. Generative Agents [19] introduced a three-part memory stream—observation, reflection, and planning—enabling simulated characters to maintain coherent behavior over extended interactions. A-MEM [22] extends this with Zettelkasten-inspired atomic notes and explicit inter-note linking. Mem0 [6] targets production deployment with a graph-based memory layer that supports multi-user, multi-session persistence. A comprehensive survey of memory mechanisms for LLM agents appears in Zhang et al. [25].

Letta [10], the productized successor to MemGPT, advances the architecture in two directions relevant to this paper. First, *context repositories* [12]: persistent, versioned stores of structured context that agents read from and write to across sessions, replacing the flat archival-storage model with typed, queryable collections. Second, *context constitution* [11]: a declarative specification of what context an agent should have access to and how it should be prioritized, analogous to a retrieval policy expressed as a configuration rather than code. These mechanisms move agent memory closer to the decision-provenance retrieval described here—context repositories provide the storage substrate, and context constitution provides a rudimentary retrieval policy. However, Letta's retrieval unit remains the context block (a text chunk with metadata), not the architectural decision with provenance, alternatives, and constitutional anchoring. The context constitution specifies *which* collections to query, not *what decision context* to retrieve for a given task.

These systems share a common retrieval unit: the conversational turn, the character observation, or the context block—closer to the architectural decision than turns or observations, but still without the explicit alternatives, validators, and constitutional anchoring that decision provenance requires. They excel at role consistency, preference tracking, and episodic recall. They are not designed for the retrieval unit that matters in long-horizon agentic engineering: the architectural decision with full provenance—which alternative was chosen, which was rejected, why, which validator enforces the choice, and which constitutional principle anchors it.

2.2 Retrieval-Augmented Code Agents

Code-RAG systems retrieve over source text to augment code generation. RepoCoder [24] iteratively retrieves similar code snippets from the repository to improve completion accuracy. CodeRAG-Bench [21] provides a systematic benchmark for retrieval-augmented code generation across documentation, API references, and code examples. Commercial systems like Cursor's codebase indexing and Sourcegraph Cody build semantic indices over entire repositories, enabling natural-language queries that return relevant code chunks. SWE-bench [9] and SWE-agent [23] evaluate agents on

real-world GitHub issues, demonstrating that retrieval over repository structure and issue context improves resolution rates.

All of these systems retrieve over *code text and identifiers*. They can answer "where is this function defined?" or "what does this module do?" but not "why was this approach chosen over the alternative?" or "which constitutional principle does this validator enforce?" The decision context—the rationale, the rejected alternatives, the cross-cutting constraints—is absent from the code itself and therefore absent from code-RAG retrieval.

2.3 Architecture Decision Records

The practice of recording architectural decisions has a long history in software engineering. Nygard's Architecture Decision Record (ADR) format [16] proposes a lightweight Markdown template—title, status, context, decision, consequences—stored alongside the codebase. The ISO/IEC/IEEE 42010 standard [8] formalizes architecture description at the organizational level. Zörner's Y-statement format [27] structures decisions as "In the context of [situation], facing [concern], we decided [option], to achieve [quality], accepting [downside]." Zimmermann et al. [26] extends ADR management to cross-project guidance with problem-space modeling.

ADRs capture decision provenance—exactly the content missing from code-RAG. However, ADR retrieval is file-path-based: decisions are found by browsing a directory, not by semantic query. There is no embedding index, no hybrid retrieval, no re-ranking by relevance to a current task. ADRs also require manual authoring; in a high-velocity workflow producing 14.7 PRs/day, the curation overhead of maintaining a complete ADR directory is prohibitive.

2.4 Long-Context Language Models as a Substitute

Gemini 1.5 [20] demonstrated effective processing of inputs up to 10M tokens in controlled settings. This has led to a common assumption that sufficiently large context windows eliminate the need for retrieval.

Empirical evidence does not support this assumption for the regime considered here. Liu et al. [14] showed that model performance degrades significantly for information placed in the middle of long contexts—the "lost in the middle" effect—even for models explicitly designed for long inputs. RULER [7] confirmed that effective context utilization drops sharply as input length grows, with most models failing to maintain claimed context lengths under adversarial probing. Li et al. [13] demonstrated that in-context learning performance degrades as the number of demonstrations grows, even when the total context fits within the window.

Beyond attention degradation, long-context processing is expensive. At the 6-week horizon of the operational regime described here, the accumulated decision context—ADRs, validator histories, constitutional principle invocations, edit-trace summaries—would consume hundreds of thousands of tokens per session start. Retrieval selects the task-relevant subset; long context pays for everything.

Synthesis. None of the four lines of work treats decision provenance as a first-class memory unit with semantic retrieval. None provides a slow-loop refresh primitive that keeps dormant-task memory current across multi-week horizons. The architecture described in this paper occupies the intersection: it adopts the provenance structure of ADRs, the semantic retrieval of code-RAG, the persistence of conversational memory systems, and the selective context loading of RAG—while adding dual-mode retrieval as a new architectural primitive.

3 Problem Formalization

Long-horizon agentic composition is a regime characterized by three measurable properties, verified in the companion paper's pilot dataset [17, §3–4]:

Horizon. Related decisions span weeks to months. The platform described here maintained active development across 105 days with a median inter-PR interval under 2 hours but architectural threads that persisted for 4–6 weeks.

Compositionality. Decision A in week 2 constrains decision B in week 6. A choice of database schema in the EDRSR import pipeline determined the parameter contract of 29 MCP tools built over the following five weeks; a constitutional principle established in week 1 governed validator behavior through week 15.

Persistence. The codebase and its associated decision graph serve as shared state between sessions, not transient context. The agent does not carry forward a dialogue history; it re-enters a changed codebase each session.

The session bootstrap problem. Each new agentic session begins by loading context. The agent consumes T input tokens of `CLAUDE.md` content and system context automatically, then performs K exploratory file reads before beginning productive work. Measured across 304 sessions from the companion paper's dataset (Section 7.2): median bootstrap cost is $T = 30{,}115$ input tokens, of which ∼17,600 (59%) is cache-creation cost for `CLAUDE.md` alone; median per-session file reads are $K = 14$ (mean 23.1). Both quantities grow approximately linearly with project age as `CLAUDE.md` accumulates conventions and the agent's file-read heuristics expand (`CLAUDE.md` growth: 217 chars/day, $R^2 = 0.87$; see Section 7.2).

The relevant subset for any given task is a small fraction of K and T. Measured across 180 sessions with at least three bootstrap reads: the median *context waste ratio*—files read but never subsequently edited—is 60%, with source-code reads wasted at 78% (Section 7.2). A task touching the billing service does not need the EDRSR import pipeline's conventions; a task modifying the frontend router does not need the PostgreSQL migration patterns. Yet the flat bootstrap loads everything, because the agent cannot predict which subset is relevant without first loading the full context.

Pull-based retrieval reduces K and T. A memory query replaces the flat bootstrap: the task description is embedded, matched against a pre-indexed memory store, and the top-k relevant entries are assembled into a compact digest. The agent starts with a focused context ($K' \ll K$, $T' \ll T$) and can retrieve additional entries on demand via MCP tool calls.

Push-based refresh keeps the substrate current. Pull-based retrieval alone fails for tasks that are dormant for weeks and then resume. During the dormant period, other tasks' decisions accumulate context that is relevant to the dormant task—schema changes, principle additions, validator updates—but no pull query fires to incorporate this context. When the dormant task resumes, the pull query retrieves stale entries from the last active period. Push-mode refresh addresses this: a scheduled process watches task activity and refreshes memory entries for dormant tasks proportional to the rate of relevant changes, so that pull queries on resumption return current context.

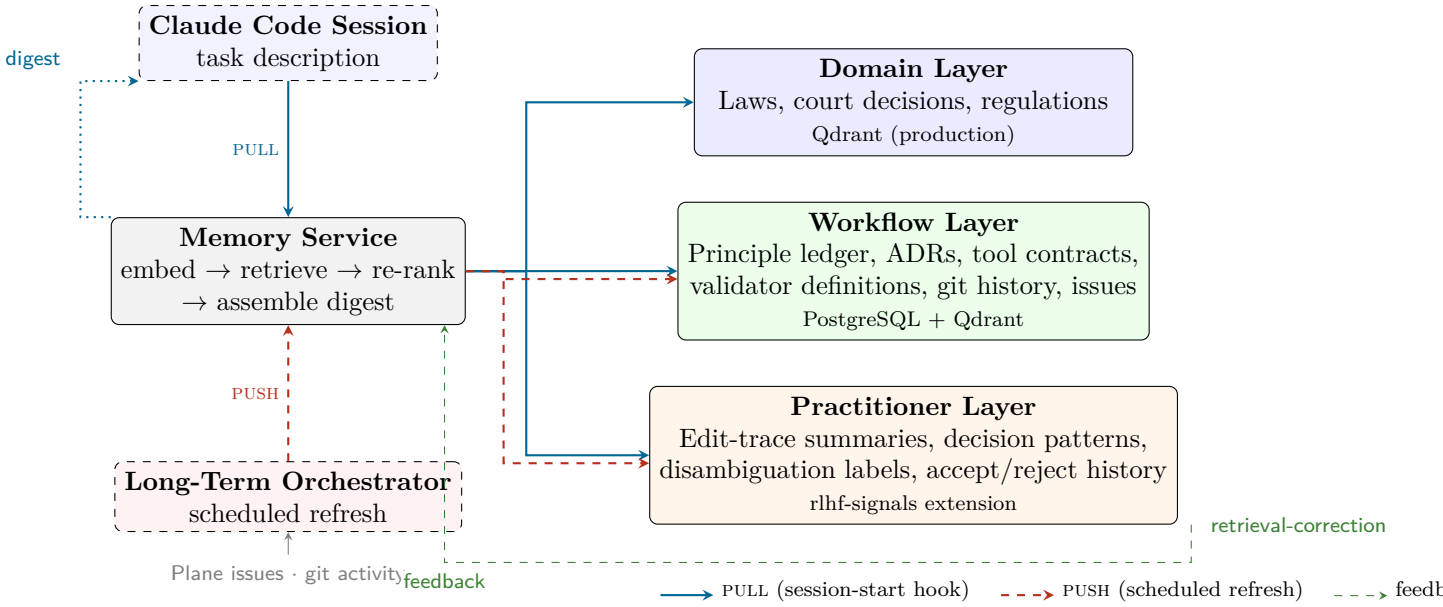

Figure 1: Three-layer workflow memory architecture with dual-mode retrieval. PULL mode (solid blue) fires at session start: the task description is embedded and used to query all three Qdrant collections in parallel; results are re-ranked and assembled into a compact digest. PUSH mode (dashed red) runs on a schedule proportional to task activity, refreshing workflow and practitioner layer entries so that pull queries return current context. The retrieval-correction feedback loop (green) detects memory entries that *should* have been retrieved but were not, feeding back into retrieval tuning and the RLHF pipeline as a process-level oversight signal.

4 Architecture

The workflow memory service decomposes into three layers, each with distinct content, retrieval semantics, and storage substrate. Figure 1 shows the overall architecture with both retrieval modes.

4.1 Domain Layer

The domain layer contains the project's operational data: laws, court decisions, regulations, and other legal corpus material. In the deployed system, this layer is already in production as multiple Qdrant vector collections served through MCP tools (semantic search, legislation retrieval, court decision lookup).

The architectural change is at the interface level. Domain queries are routed through the memory service rather than directly to individual MCP tools. This indirection provides three capabilities that direct tool calls lack: (1) unified logging of all retrieval events, enabling retrieval-miss detection (Section 5.4); (2) miss-rate measurement per collection, informing re-indexing priorities; and (3) a uniform response format across all three layers, simplifying the digest assembly step.

The domain layer uses pure semantic retrieval: the query embedding is matched against document-chunk embeddings via cosine similarity, with optional metadata filtering (jurisdiction, date range, document type).

4.2 Workflow Layer

The workflow layer contains the project's decision context—the provenance that is absent from both code text and domain data. Six sources populate this layer, listed in priority order:

1. **Principle ledger.** The highest-priority source. Each entry is a principle that the model violated and was corrected on—the hardest signal in the system. Entries are compact (typically one sentence plus metadata), referenced by validators (so retrievals are dense), and anchored to the constitutional framework described in the companion paper [17, §5]. Schema: `id`, principle statement, severity (critical/warning/info), validator references, related ADR IDs, edit-trace links, embedding vector.
2. **Validator definitions.** Source code of validation functions, parsed and embedded with docstrings, type signatures, and test examples. Validators encode the operational expression of constitutional principles; retrieving the validator alongside the principle provides the agent with both the "what" and the "how."
3. **MCP tool contracts.** For each of the 70+ tools: name, description, parameter schema, example inputs and outputs. Embedded as structured text. Tool contracts evolve frequently in early development; the embedding pipeline re-indexes on each build.
4. **ADR-style decision records.** One Markdown file per significant architectural decision, following Nygard's format [16]: context, decision, alternatives considered, consequences, related constitutional principles. Manually authored in Phase 1; automatic generation from PR descriptions is planned for Phase 2.
5. **Git history with semantic enrichment.** For each merged PR: title, description, semantic class (from the companion paper's taxonomy), affected components, related ADR IDs. Provides temporal context for when and why code changed.
6. **Issue tracker state.** Plane[1] issues with state transitions, assignees, and linked PRs. Embedded with structured metadata to enable queries like "what decisions were made on the billing migration?"

The workflow layer uses hybrid retrieval: semantic similarity against embedded text, combined with structured filters on component, layer, severity, and temporal range.

4.3 Practitioner Layer

The practitioner layer contains the practitioner's decision patterns, extracted from the same edit-trace pipeline described in the companion paper [17, §3]. This layer is the cross-reference point: the infrastructure that enables one practitioner to ship 1,547 PRs in 105 days *generates* the preference signal the RLHF paper analyzes.

The layer never embeds raw content. Following the privacy boundary established in the companion paper [17, §6], only structured edit summaries are indexed: edit class, semantic category, affected component, outcome (accepted/rejected/modified), and a one-sentence rationale generated by a nightly summarization job. This preserves the separation between the practitioner's operational data (which may contain client-specific information) and the memory substrate (which contains only structured abstractions).

Retrieval is hybrid: semantic similarity combined with temporal and outcome-conditioned filtering. A query about authentication patterns retrieves edit summaries where the practitioner corrected authentication-related code, ordered by recency and weighted by outcome confidence.

[1] https://plane.so

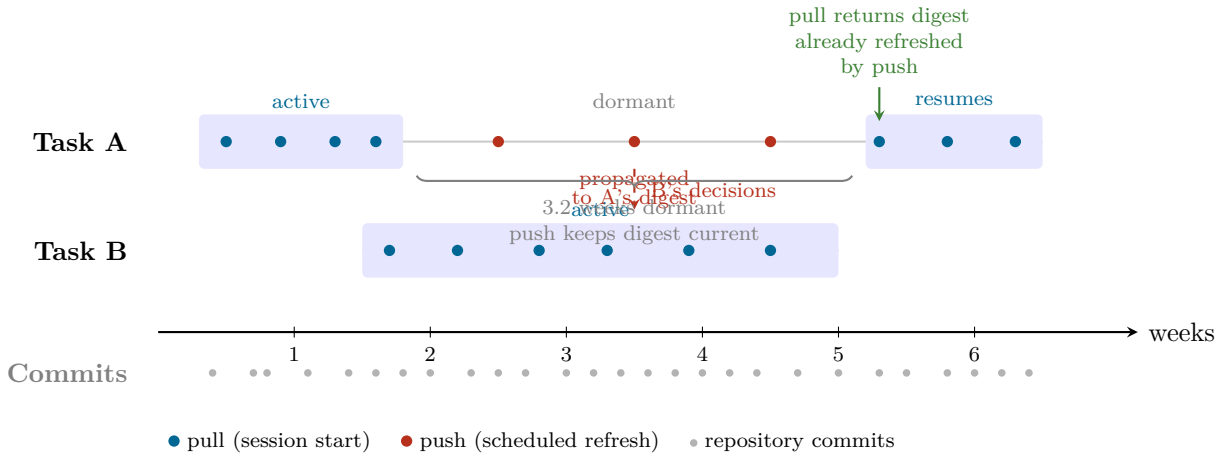

Figure 2: Dual-mode retrieval over a six-week horizon with two concurrent tasks. Task A is active in weeks 1–2, then dormant while Task B runs in weeks 2–5. During dormancy, push-mode refresh propagates decisions from Task B into Task A's memory digest at a frequency proportional to commit activity. When Task A resumes in week 5, the pull query returns context that is already current — no expensive re-derivation required. Without push mode, the week-5 pull would return a stale digest from week 2, missing three weeks of relevant architectural decisions.

Phase 0: Prompt-Commit Bridge. Before the full practitioner layer, a minimal data collector is deployed: a `UserPromptSubmit` hook in Claude Code that creates an orphan commit in a bare git repository for every prompt. Each record contains the prompt text, timestamp, session ID, repository, branch, and permission mode. Over 4–6 weeks of passive collection, this yields a natural-text corpus for disambiguation labels, topic-frequency distributions, and cross-project switching patterns—the raw material for the practitioner layer's embedding pipeline. Performance overhead is 15ms per capture, imperceptible within Claude Code's 5-second hook timeout.

5 Dual-Mode Retrieval

The dual-mode retrieval architecture is the central contribution: pull-based retrieval for active sessions, push-based refresh for dormant tasks, operating as complementary mechanisms rather than alternatives. Figure 2 illustrates the interaction between both modes across a six-week horizon.

5.1 Pull Mode

Pull mode fires at session start via a Claude Code session-start hook. The process:

1. The practitioner provides a task description (or the system infers it from the active Plane issue).
2. The task description is embedded using the same model used for memory indexing (text-embedding-3-small or a locally hosted alternative such as bge-large-en-v1.5).
3. Three parallel Qdrant queries execute against the domain, workflow, and practitioner collections, each with collection-specific filters (jurisdiction constraints for domain, component/severity for workflow, recency weighting for practitioner).
4. Results from all three collections are merged and re-ranked using a cross-encoder [15] or LLM-as-reranker, producing a unified relevance ordering.

5. A greedy fill algorithm assembles the top-ranked entries into a token-budgeted digest, with a diversity constraint that ensures representation from all three layers.

Latency target: < 500ms wall time for the full pipeline (embed, query, re-rank, assemble). This is a design target, not a measurement; empirical latency will be reported after deployment.

5.2 Push Mode

Push mode runs as a scheduled orchestrator—a separate Docker container with a Bedrock Claude Haiku summarization pipeline. The orchestrator watches Plane tasks tagged `LONG-TERM` and refreshes their memory entries proportional to task activity.

The refresh frequency is a function of three signals:

$$f_{\text{refresh}}(\text{task}) = g(\text{commits-touching-task},\ \text{comments-on-issue},\ \text{time-since-last-pull}) \tag{1}$$

where g is a monotonically increasing function of all three arguments, clamped to a maximum of one refresh per 24 hours and a minimum of one per 7 days for any task tagged `LONG-TERM`.

Each refresh produces three outputs:

- An **executive summary** of changes relevant to the task since the last refresh, generated by Bedrock Claude Haiku from the diff of relevant commits and issue comments.
- A **tool lineage delta**: new tools created, tools modified, tools deprecated since the last refresh, with parameter-schema diffs.
- **Open questions**: unresolved issues or decision points flagged by the summarizer that may require practitioner input on task resumption.

The executive summary and tool lineage delta are embedded and stored in the workflow layer's Qdrant collection. On the next pull query for the refreshed task, these entries surface alongside the original memory entries, providing the session with current context.

5.3 Why Both Modes Are Necessary

Pull-only retrieval fails when a task is dormant for weeks and then resumes. During the dormant period, other tasks' decisions accumulate relevant context: schema changes that affect the dormant task's data model, new constitutional principles that constrain its implementation, validator updates that redefine correctness criteria. None of this context is incorporated into the dormant task's memory entries, because no pull query fires during dormancy. When the task resumes, the pull query retrieves entries from the last active period—stale by weeks. Re-deriving the current state requires expensive exploratory file reads, manual context assembly, and risk of missing cross-task dependencies.

Push-only refresh, conversely, is wasteful for active tasks. A task with multiple sessions per day does not need pre-computed digests; each session's pull query returns current results because the memory entries were recently indexed. Push refresh for active tasks duplicates the work that pull already performs, consuming Bedrock API credits without improving retrieval quality.

Dual-mode retrieval allocates computational effort where it provides the most value: pull for fast loops (active sessions), push for slow loops (dormant tasks). The push mode is what makes 6-week horizons tractable; without it, retrieval over multi-week dormancy periods returns stale or absent context.

5.4 Retrieval-Correction Edits

Retrieval-correction edits operationalize a specific oversight gap: the model produced an output that the practitioner corrected because the model's available context was missing information the practitioner had. Definition: an edit where the model's output would have been correct given access to context X, but X was not present in the session's memory digest. This edit class extends the six-class taxonomy from the companion paper [17, §4], but is structurally distinct: the other six classes capture *what* the practitioner changed; retrieval-correction captures *why* the change was avoidable. It is therefore a process-level oversight signal rather than a preference signal in the standard RLHF sense.

Detection proceeds via a post-session reconciliation job:

1. After the session closes, the reconciliation job collects all edits made by the practitioner.
2. For each substantive edit (edit distance > 0.3 in the companion paper's normalized metric), the job embeds the edit context and queries the memory store for entries that are semantically similar to the edit but were *not* retrieved in the session's digest.
3. Candidate retrieval-correction pairs (edit, unretrieved-but-relevant memory entry) are flagged for practitioner review.
4. Confirmed pairs are labeled as retrieval-correction edits and enter the alignment pipeline as oversight-derived preferences: the "chosen" response incorporates the missing context; the "rejected" response is the original model output that lacked it.

This closes a feedback loop between the memory layer and the alignment pipeline that is structurally different from outcome-level RLHF. Outcome-level supervision answers "did this run succeed?" once per session. Retrieval-correction supervision answers "was this specific output reachable from this specific context?" per edit. The signal density per unit of agent activity is therefore higher, and the signal is invariant to session length. Two consequences follow. First, as the memory layer matures, retrieval-correction edit rate becomes a measurable proxy for memory effectiveness—an internal evaluation metric that does not require waiting for downstream outcomes. Second, retrieval-correction edits encode a failure mode (capability present, context absent) that is distinct from capability gaps (capability absent, context irrelevant). Distinguishing these failure modes is a long-standing open problem in scalable oversight [5]; the memory layer provides one concrete instantiation.

6 Implementation

The memory service is implemented in TypeScript, consistent with the existing Express-based MCP backend [1] described in the companion paper [17, §7]. The technology stack reuses production infrastructure: Qdrant for vector storage, PostgreSQL for structured metadata (new `workflow_memory_*` tables), Redis for caching, and AWS Bedrock Claude Haiku for summarization steps.

6.1 Memory Query Tool

A single MCP tool, `workflow_memory_query`, serves as the primary retrieval interface. Parameters:

- `task_description` (string, required): natural-language description of the current task.
- `scope` (enum: `all` | `domain` | `workflow` | `practitioner`): which layers to query. Default: `all`.
- `token_budget` (integer, default 8,000): maximum tokens in the assembled digest.
- `filters` (object, optional): structured filters (component, severity, date range).

The tool returns a structured prose digest with section headers per layer, relevance scores per entry, and source provenance links. The digest format is designed for direct injection into the agent's

context window without post-processing.

6.2 Long-Term Task Orchestrator

The push-mode orchestrator runs as a separate Docker container with its own cron schedule. It queries the Plane API for tasks tagged `LONG-TERM`, computes refresh priority per Equation 1, and dispatches summarization jobs to Bedrock. Outputs (executive summaries, tool lineage deltas, open questions) are embedded and stored in the workflow layer's Qdrant collection with a `source: push-refresh` metadata tag.

All seven phases (0, 1.0–1.5) have been implemented and deployed. The complete implementation spans three PostgreSQL migrations (`144`–`146`), three Qdrant vector collections (`wm_domain`, `wm_workflow`, `wm_practitioner`), and six MCP tools (`workflow_memory_query`, `workflow_memory_ingest`, `workflow_memory_stats`, `workflow_memory_reconcile`, `workflow_memory_push_sync_tasks`, `workflow_memory`
The phase schedule and implementation details are available in the supplementary materials.[2]

7 Evaluation

7.1 Metrics

Four metrics operationalize the architecture's success criteria, each paired with a baseline measurement method and a target.

A fifth metric—the retrieval-correction edit rate over time—is qualitative in Phase 1. As the practitioner layer matures, the rate of retrieval-correction edits (Section 5.4) should decrease. The expected trajectory is a declining curve over the first 6–8 weeks of deployment, flattening as the memory layer covers the most frequently needed context.

7.2 Preliminary Results

Three baseline measurements are available at submission time, derived from the companion paper's transcript dataset and from the project's git history.

Bootstrap cost (Experiment 1, N=304 sessions). Parsing Claude Code API transcripts from the companion dataset, the median first-call input token count is 30,115 (mean 29,693; σ=7,594; P10/P90: 18,540/41,774). Of these, ∼17,600 tokens (59%) are cache-creation cost for `CLAUDE.md` and system context—loaded unconditionally on every session regardless of task. Per-session file reads (Read tool calls) average 23.1 (median 14, P90: 61), but 72% of sessions have zero Read calls in the bootstrap phase; most sessions open with shell commands (`git status`, `ls`) rather than file reads. The memory layer's primary reduction target is therefore the automatic context cost (T), not the file-read count.

CLAUDE.md growth (Experiment 2, 25 commits over 85 days). The project's `CLAUDE.md` grew from 4,099 to 24,148 characters (152 to 474 lines) across 25 commits over 85 days (January 17 to April 12, 2026—a subset of the 105-day operational period documented in the companion paper). A linear regression of character count on days elapsed yields a slope of 216.7 chars/day with $R^2 = 0.87$, confirming the approximately linear growth claimed in Section 1. Line count is noisier ($R^2 = 0.58$) due to periodic reformatting that changes density without changing content volume.

[2] Phase schedule: https://github.com/overthelex/SecondLayer/blob/main/docs/workflow-memory-phases.md

Context waste ratio (Experiment 3, N=180 sessions). Across sessions with at least three file reads in the bootstrap phase (first 50 turns), the median context waste ratio—files read but never subsequently edited in the session—is 60% (mean 56.7%; σ=26.3%). The distribution is right-skewed: 66% of sessions waste more than half their bootstrap reads; only 15% waste less than 30%. Source-code reads are wasted at 78%, while memory files (`.claude/memory/`) are wasted at 66.5%, suggesting that structured memory retrieval is more targeted than exploratory file reads. Waste ratio correlates negatively with session length: short sessions (5–10 turns) waste 68.5%; longer sessions (>50 turns) waste 51.1%. This metric is a conservative proxy: a file may be read for context without subsequent editing, in which case it counts as "wasted" even when its content informed the agent's reasoning. The 60% baseline therefore likely overstates true waste; the gap between read-and-edited and read-and-used is left for future work.

Phase 0 (prompt-commit bridge) has been deployed since May 9, 2026 with negligible performance overhead (15ms per capture); the corpus contains 533 prompts across 13 qualifying sessions ($\geq$5 prompts each) at time of writing.

Deployment state (all phases complete). All seven phases (0, 1.0–1.5) are deployed on the local environment as of May 13, 2026. The memory layer contains 187 entries across all three layers: 170 domain principles (140 from `CLAUDE.md` sections, 23 from merged PR descriptions, 7 from design documents), 4 workflow patterns (push-refresh summaries generated via Bedrock Claude Haiku), and 13 practitioner session summaries (covering 533 captured prompts across 13 qualifying sessions). All embeddings use Amazon Titan Embed Text v2 (1024 dimensions) via AWS Bedrock.

Principle retrieval quality was validated with a round-trip test: the query "how do we deploy to production" returned 5 relevant principles (blue-green deployment, CI/CD automation, no manual containers, container health checks, Docker rebuild before testing) with cosine similarity scores ranging from 0.522 to 0.604.

The reconciliation tool (Phase 1.4) was tested on a session that modified 3 files (`deployment/docker-compose.lo` `mcp_backend/src/services/workflow-memory-service.ts`, `mcp_backend/src/migrations/145*.sql`). Of the 170 principles, 113 were identified as relevant by tag-based and keyword-based matching; the session had retrieved only 1 principle, yielding precision = 1.0 and recall = 0.009. The low recall is expected: this session had only one `workflow_memory_query` call. The 112 missed principles were correctly flagged as retrieval-correction candidates.

Push-mode (Phase 1.5) has been deployed with the Plane task watcher, LLM-based summarization (Bedrock Claude Haiku), and tool lineage delta detection. Four active tasks are on the watchlist; the initial tool lineage snapshot covers 22 registered tools. The first push-refresh cycle completed successfully, generating 4 workflow patterns with executive summaries and open questions at a total LLM cost of $0.005.

Full A/B evaluation of bootstrap reduction (memory-enabled vs. memory-disabled sessions) requires several weeks of operational data and will be reported in a subsequent revision.

7.3 Threats to Validity

Single project, single practitioner. The architecture is designed and evaluated in the context of one project (LEX AI) and one practitioner. This is the same constraint as the companion paper [17, §8]. Mitigation: the architecture is project-agnostic—the three-layer decomposition and dual-mode retrieval impose no assumptions about the domain, the programming language, or the practitioner's expertise. Generalization claims are deferred to Phase 2, which introduces a multi-practitioner cohort.

Confounding with codebase maturity. Bootstrap reduction could result from caching effects, improved `CLAUDE.md` curation, or the natural stabilization of a maturing codebase rather than from the memory layer itself. Control: A/B evaluation by toggling the memory layer at session start. Sessions with the memory layer enabled are compared against sessions without it on the same task distribution during the same time period.

Principle ledger quality. In Phase 1, the principle ledger is human-curated. Retrieval accuracy is therefore bounded by curation quality, not retrieval quality. If the registry is incomplete or contains imprecise principle statements, the retrieval system will faithfully return imprecise results. This limitation is acknowledged and addressed in Phase 2 by introducing semi-automatic principle extraction from edit-traces.

8 Discussion

Memory layer as RLHF substrate. The same infrastructure that enables one practitioner to ship 1,547 PRs in 105 days *generates* the preference signal the companion paper analyzes. Memory and preference data are two views of the same long-horizon work: a retrieval-correction edit is simultaneously a memory-layer failure (the system did not surface relevant context) and an alignment signal (the practitioner corrected the model's output, producing a preference pair). This duality is not incidental—it is the architectural thesis. The memory layer does not merely consume alignment data; it produces it.

Memory as scalable oversight surface. The memory layer is dual-purpose: it serves the agent during execution and generates oversight signal during reconciliation. As agent autonomy grows—from short interactive sessions to multi-hour autonomous runs to multi-day autonomous projects of the kind documented in Anthropic's long-running scientific computing work [4]—the ratio of practitioner-observable outcomes to agent-internal decisions falls. Outcome-level supervision becomes a low-bandwidth channel through which any oversight signal must pass.

The memory layer changes this. Every retrieval is an observable event with a known context window, a known retrieval result, and a subsequent edit-trace. Reconciliation between retrieved context and editing behavior generates oversight signal at the granularity of individual edits, not individual sessions. The signal is not free—reconciliation requires practitioner confirmation of candidate retrieval-correction pairs—but the practitioner's marginal cost per signal is bounded by the time to confirm a flagged pair, not the time to evaluate an entire session.

This positions the memory layer as one concrete answer to the question posed in *Recommendations for Technical AI Safety Research Directions* [2]: how to design oversight mechanisms whose signal scales with the systems they oversee. The architecture does not solve scalable oversight in general. It demonstrates that for one operational regime—long-horizon agentic coding by a small number of practitioners—the memory infrastructure that makes the regime tractable also produces the signal needed to oversee it.

Generalization to teams. The architecture as described serves a single practitioner. Multi-practitioner deployment introduces three challenges not addressed here: (1) multi-writer principle ledger with conflict resolution on principle updates; (2) per-practitioner views of the practitioner layer, since different practitioners have different correction patterns; and (3) access control on memory entries that may contain project-specific context. These are Phase 2 concerns. The three-layer decomposition is designed to accommodate them—the domain layer is already shared, the

workflow layer supports scoped views via metadata filtering, and the practitioner layer is inherently per-practitioner.

What this is not. The workflow memory architecture is not a replacement for long-context models; it is a retrieval substrate that reduces the context a long-context model must process. It is not a knowledge graph: there is no formal ontology, no RDF triples, no SPARQL endpoint. It is not an enterprise knowledge management system: it has no document lifecycle, no approval workflows, no compliance metadata. It is a retrieval substrate optimized for one operational regime—long-horizon agentic composition by a small number of practitioners—and its design decisions reflect that specificity.

9 Limitations

- **Single project, single practitioner.** All design and evaluation is within one legal-technology project built by one practitioner. Generalization to other domains, other languages, or multi-developer teams is not demonstrated.
- **Manual ADR curation.** In Phase 1, architecture decision records are manually authored. The curation burden is non-trivial at 14.7 PRs/day; ADR coverage is expected to be incomplete during the evaluation period.
- **Batch re-indexing.** Memory entries are re-indexed nightly, not in real time. Changes made during a session are not reflected in the memory layer until the next indexing run. This creates a window where pull queries may return slightly stale results for very recent changes.
- **No multi-practitioner support.** The architecture does not address multi-writer conflicts in the principle ledger, per-practitioner memory views, or cross-practitioner preference aggregation.
- **Retrieval-correction detection is approximate.** The post-session reconciliation job uses semantic similarity to identify candidate retrieval-correction pairs. False positives (flagged pairs where the missing context would not have changed the model's output) are filtered by practitioner review, but the detection precision is not yet measured.
- **Push-mode cost.** Each push refresh consumes Bedrock API credits for summarization. At scale, the cost of maintaining current digests for many dormant tasks may become significant. The current design mitigates this with frequency clamping (maximum one refresh per 24 hours per task), but cost-optimal refresh scheduling is not formalized.
- **Oversight-signal density is not yet quantified.** The claim that retrieval-correction edits constitute a denser oversight signal than outcome-level supervision is structurally argued but not measured. Quantifying signal density (retrieval-correction edits per practitioner-minute vs. outcome labels per practitioner-minute) requires the full deployment to run for several weeks; this measurement is part of the evaluation plan.

10 Conclusion

This paper presented a workflow memory architecture for long-horizon agentic composition, addressing the session bootstrap problem that bottlenecks multi-week LLM-assisted software engineering. Three contributions were made. First, a three-layer memory decomposition—domain, workflow, and practitioner—where each layer has distinct content, retrieval semantics, and storage substrate, and the retrieval unit is the architectural decision rather than the code chunk or dialogue turn. Second, dual-mode retrieval as a first-class primitive: pull mode for active sessions, push mode for dormant tasks, with push-mode refresh keeping multi-week horizons tractable by maintaining

current memory entries without requiring active sessions. Third, retrieval-correction edits as a process-level oversight signal that scales with agent autonomy, closing the feedback loop between the memory layer and the alignment pipeline described in the companion paper [17].

The architecture is deployed incrementally on a production legal-technology platform. All seven phases (0, 1.0–1.5) have been implemented and deployed on the local environment as of May 13, 2026. Baseline measurements from 304 sessions confirm a median bootstrap cost of 30,115 input tokens and a context waste ratio of 60%; the memory layer targets $\leq$10K tokens and $\leq$20% waste, with $\geq$0.80 top-1 cosine similarity on principle ledger entries.

The source code and implementation notes are available at https://github.com/overthelex/SecondLayer.

References

[1] Anthropic. Model context protocol specification. https://modelcontextprotocol.io, 2024. Open protocol for connecting LLM applications to external data sources and tools.

[2] Anthropic Alignment Science Team. Recommendations for technical AI safety research directions. Alignment Science Blog, Anthropic, 2025. URL https://alignment.anthropic.com/2025/recommended-directions/. Accessed: 2026-05-10.

[3] Anthropic Engineering. Effective harnesses for long-running agents. Anthropic Engineering Blog, 2026. URL https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents. Accessed: 2026-05-10.

[4] Anthropic Research. Long-running Claude for scientific computing. Anthropic Research Blog, 2026. URL https://www.anthropic.com/research/long-running-Claude. Accessed: 2026-05-10.

[5] Samuel R. Bowman, Jeeyoon Hyun, Ethan Perez, Edwin Chen, Craig Pettit, Scott Heiner, et al. Measuring progress on scalable oversight for large language models. *arXiv preprint arXiv:2211.03540*, 2022. URL https://arxiv.org/abs/2211.03540.

[6] Prateek Chhikara, Deshraj Khetan, et al. Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025. URL https://arxiv.org/abs/2504.19413.

[7] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024. URL https://arxiv.org/abs/2404.06654.

[8] ISO/IEC/IEEE. ISO/IEC/IEEE 42010:2011 — systems and software engineering — architecture description. International standard, International Organization for Standardization, 2011.

[9] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2024. URL https://arxiv.org/abs/2310.06770.

[10] Letta. Letta Code: A memory-first coding agent. Letta Blog, 2025. URL https://www.letta.com/blog/letta-code. Accessed: 2026-05-10.

[11] Letta. Context constitution. Letta Blog, 2026. URL https://www.letta.com/blog/context-constitution. Accessed: 2026-05-10.

[12] Letta. Introducing context repositories: Git-based memory for coding agents. Letta Blog, 2026. URL https://www.letta.com/blog/context-repositories. Accessed: 2026-05-10.

[13] Tianle Li, Ge Ge, Aakanksha Joshi, Yueqi Peng, Xiang Yu, et al. Long-context LLMs struggle with long in-context learning. *arXiv preprint arXiv:2404.02060*, 2024. URL https://arxiv.org/abs/2404.02060.

[14] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. URL https://arxiv.org/abs/2307.03172.

[15] Rodrigo Nogueira and Kyunghyun Cho. Passage re-ranking with BERT. *arXiv preprint arXiv:1901.04085*, 2019. URL https://arxiv.org/abs/1901.04085.

[16] Michael Nygard. Documenting architecture decisions. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions, 2011. Accessed: 2026-05-10.

[17] Vladimir Ovcharov. Edit-trace oversight: Scalable alignment signal from agentic workflows. 2026. Manuscript in preparation. Companion paper. 1,547 PRs, 30,510 edit-traces, 105 days.

[18] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

[19] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023. URL https://arxiv.org/abs/2304.03442.

[20] Machel Reid, Nikolay Savinov, Denis Teber, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024. URL https://arxiv.org/abs/2403.05530.

[21] Zora Zhiruo Wang, Akari Shi, Jeremiah Milbauer Yang, Daniel Fried, and Graham Neubig. CodeRAG-Bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*, 2024. URL https://arxiv.org/abs/2406.14497.

[22] Wujiang Xu, Zujie Liang, Jingyang Mei, Hang Xiao, Jianping Li, et al. A-MEM: Agentic memory for LLM agents. *arXiv preprint arXiv:2411.07559*, 2024. URL https://arxiv.org/abs/2411.07559.

[23] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Liber, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793.

[24] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Chen, and Wenqiang Chen. RepoCoder: Repository-level code completion through

iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023. URL https://arxiv.org/abs/2303.12570.

[25] Zeyu Zhang, Xiaohe Bo Zhang, Chen Jiang, Bin Liu, Jiaqi Fan, Jiwen Zhu, et al. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*, 2024. URL https://arxiv.org/abs/2404.13501.

[26] Olaf Zimmermann, Lukas Wegmann, Cesare Pautasso, and Oliver Kopp. Architectural decision guidance across projects: Problem space modeling, decision backlog management and cloud computing knowledge. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture*, 2015.

[27] Stefan Zörner. *Softwarearchitekturen dokumentieren und kommunizieren*. Hanser, 2nd edition, 2015. Introduces the Y-statement format for architectural decisions.

Metric	Measurement method	Baseline	Target
Bootstrap token cost	Input tokens at first API call (includes `CLAUDE.md` cache); N=304 sessions	30,115 (median)	≤10K
Context waste ratio	Files read but never edited / total files read; N=180 sessions with ≥3 reads	60% (median)	≤20%
Principle retrieval accuracy	Held-out set of principle ledger entries; queries derived from edit-traces that originally triggered each principle	0.604 (top-1 cosine similarity)	≥0.80 (top-1 cosine similarity)
Long-term task refresh latency	Time from relevant commit to updated digest entry	deployed	≤24h

Table 1: Evaluation metrics with measured baselines and targets. Bootstrap baselines are measured from 304 Claude Code sessions in the companion paper's transcript dataset.