From Observed Reasoning to Stable Skills: A Memory Substrate for Skill Graduation

Cognitive Memory Manager: Capturing, Validating, and Promoting Procedural Knowledge from Coding Agent Sessions

Sazan Khalid
Georgetown University
sk2153@georgetown.edu

Amit Arora
Amazon Web Services
amiarora@amazon.com

Abstract

The Agent Skills format gives LLM-based agents portable procedural knowledge through `SKILL.md` files. Skills are powerful when curated and harmful when misauthored: a recent benchmark reports a 16.2-point average pass-rate improvement for curated Skills, yet self-generated Skills degrade performance in 31% of evaluated tasks. We argue that this gap reflects a missing observation layer: Skills are authored without empirical evidence about which procedural patterns actually recur and pay off across real agent sessions. We present the Cognitive Memory Manager (CMM), a memory system that observes coding agent execution, extracts reasoning structure as a directed acyclic graph, accumulates patterns across sessions and developers under a human-approval gate, and serves them to future sessions. On top of this substrate we propose and implement a Skill graduation mechanism: criteria under which a memory pattern, once validated by repeated observation and successful retrieval, is promoted to a stable `SKILL.md`. We evaluate CMM with a case study A/B comparison on a real bug in a mid-sized enterprise open-source codebase. With CMM-derived knowledge loaded at session start, an agent reduced assistant messages by 61% (119→46), files modified by 71% (7→2), and reached the correct root cause 11 messages earlier than the baseline; limitations of this single-run evaluation are discussed in Section 11. The system, including extraction pipeline, distributed sync layer, graduation command, and 216-test harness, is released under Apache 2.0.

Keywords: agent skills, procedural knowledge, memory systems, trajectory mining, coding agents, LLM observability, SKILL.md graduation

ACM Reference Format:
Sazan Khalid and Amit Arora. 2026. From Observed Reasoning to Stable Skills: A Memory Substrate for Skill Graduation: Cognitive Memory Manager: Capturing, Validating, and Promoting Procedural Knowledge from Coding Agent Sessions. In *Proceedings of The First Workshop on Agent Skills (Agent Skills '26).* ACM, New York, NY, USA, 8 pages.

Agent Skills '26, San Jose, CA
2026.

1 Introduction

The Agent Skills format [3] packages procedural knowledge for LLM agents in portable `SKILL.md` files, adopted across Claude Code, Gemini CLI, and 30+ agent platforms. Recent benchmarks confirm that Skills work: SkillsBench [2] reports a 16.2-point average pass-rate improvement for curated Skills. The same benchmark also reports a sobering result: self-generated Skills regress performance in nearly a third of evaluated tasks. A separate audit found 37% of community Skills contain security flaws [1]. The community now has a clear technical problem: *Skills are valuable, but their authoring is unreliable.*

We argue that the gap between curated and self-generated Skills reflects a missing component of the Skill lifecycle: *observation.* Curated Skills are written by experts who have already lived through the patterns they encode. Self-generated Skills are produced by an agent introspecting on a single session; the agent confidently asserts procedural knowledge it has no evidence will generalize. The bridge between these regimes is empirical observation across many sessions, with a mechanism for promoting patterns that demonstrably recur and pay off.

This paper presents that bridge. The Cognitive Memory Manager (CMM) is a memory substrate that observes coding agent execution, extracts reasoning structure from session transcripts, and accumulates validated patterns across sessions and developers. CMM is not itself a Skill; it is the system from which good Skills can be derived. We then propose and implement a graduation mechanism that promotes high-confidence, frequently retrieved memory patterns into stable `SKILL.md` entries.

The substrate has three layers. First, an extraction pipeline captures each coding session as a typed reasoning DAG, hypotheses, investigations, dead ends, pivots, solutions, with two cost tiers (heuristic and LLM-based). Second, a consolidation pipeline groups related reasoning across sessions into ranked pitfalls, architectural insights, and diagnostic strategies, using confidence-prioritized cluster sampling so that the most reliable signal drives consolidation. Third, a distributed sync layer with a human approval gate prevents hallucinated patterns from propagating to a team.

The graduation mechanism specifies when a memory pattern becomes a Skill. We propose three criteria: cross-session frequency (the pattern recurs across multiple independent sessions), retrieval validation (it is retrieved and used during real agent runs), and human ratification (it passes the approval gate). Patterns that meet all three are promoted; patterns that fail to recur or are never retrieved decay in confidence and are eventually retired. This produces a Skill set that is empirically grounded rather than speculative.

Contributions.

1. A reasoning DAG model for coding agent sessions that captures the structure of agent decision-making, with a two-tier extraction pipeline (warm heuristic, cold LLM) optimized via token-budget context windowing at 45% fill (Sections 3–4).
2. A consolidation algorithm that groups reasoning across sessions and produces actionable, evidence-grounded summaries via confidence-prioritized cluster sampling (Section 5).
3. A Skill graduation mechanism specifying empirical criteria under which observed memory patterns are promoted to stable `SKILL.md` entries, with the inverse path for patterns that fail to validate, and an implementation as the `cmm graduate` command (Section 8).
4. A distributed sync layer with human approval that enables team-scale knowledge sharing while preventing hallucinated content from propagating (Section 7).
5. A case study A/B evaluation on a real bug in a mid-sized enterprise codebase showing 61% reduction in messages-to-solution and 71% reduction in files modified when the agent loads CMM-derived knowledge at session start (Section 9).

The system is released open source under Apache 2.0 with 216 tests covering the extraction pipeline, storage layer, sync semantics, review workflow, confidence decay, and graduation command.

2 Related Work

Agent Skills and procedural knowledge. The `SKILL.md` specification [3] defines a markdown-based format for distributing procedural knowledge to LLM agents. SkillsBench [2] benchmarks Skills as first-class artifacts and provides the 16.2-point and 31%-regression numbers we cite. The 3,984-Skill audit [1] provides the security-flaw rate. We position CMM relative to these by addressing the authoring gap they identify: Skills work, manual authoring does not scale, and self-generated Skills are unreliable.

Agent memory systems. Mem0 [8], Letta [9], and Cognee [12] provide general-purpose memory for LLM agents through SaaS APIs and frameworks. They store conversational facts and entities. CMM differs in three ways: it targets coding-specific reasoning rather than general assistant memory; it extracts the structure of reasoning (typed DAG nodes with causal edges) rather than flat facts; and it is designed as a substrate from which stable Skills are derived, with explicit graduation criteria.

Coding agent memory. Claude Code shipped Auto Memory in February 2026, in which the agent maintains a `MEMORY.md` of build commands and preferences, with consolidation via Auto Dream [4]. Cursor provides similar mechanisms via `.cursorrules` [6]. These are single-developer and store agent self-reports. They do not extract reasoning structure from session transcripts and do not support team-scale knowledge sharing or graduation to stable artifacts.

Trajectory-based learning. ExpeL [13] extracts insights from agent trajectories using string heuristics. Reflexion [11] uses verbal self-reflection within a single task. CMM differs by producing structured DAGs rather than flat insight lists, by operating across sessions and developers rather than within a trajectory, and by terminating in a portable artifact (`SKILL.md`) consumable by any compliant agent platform.

Long-context degradation. Our extraction strategy depends on results from the long-context literature. Chroma's 2025 study [5] shows uniform performance degradation across 18 frontier models with increasing context length. Liu et al. [7] demonstrate the "lost in the middle" effect with 30+% accuracy drops for mid-context information. Paulsen [10] introduces the Maximum Effective Context Window (MECW) and shows it is far below advertised maximums. We use these results to calibrate our 45% context fill ratio for cold-tier extraction.

3 The Reasoning Graph Model

CMM treats every coding session as a directed acyclic graph $G = (V, E)$, where each $v \in V$ is a typed reasoning node and each $e \in E$ is a causal relationship. The graph flows forward in transcript time: when an agent revisits a topic, the revisit produces a new node rather than a back-edge.

3.1 Node Types

Seven node types capture the structure of coding agent reasoning:

- `HYPOTHESIS`: the agent forms a theory ("maybe the issue is in the routing factory").
- `INVESTIGATION`: the agent gathers evidence (file reads, command executions).
- `DISCOVERY`: an unexpected finding.
- `PIVOT`: an explicit change of approach.
- `DEAD_END`: a failed approach (error, test failure, futile command).
- `SOLUTION`: a working resolution.
- `CONTEXT_LOAD`: initial reading to build understanding.

The set is intentionally small. `DEAD_END`, `PIVOT`, and `SOLUTION` carry the strongest signal for downstream consolidation: they encode what to avoid, when to change direction, and

what works. INVESTIGATION nodes are abundant but contribute primarily through their relationships with adjacent high-signal nodes.

3.2 Node Schema

Each node stores six fields. summary (1–2 sentences) is the natural-language description used for embedding and retrieval. evidence grounds the node in a specific transcript fragment (an error message, a command output, or a quoted assistant statement), which downstream stages use to ensure that consolidated Skills remain anchored in observed reality rather than LLM elaboration. message_range preserves the source position. confidence is the extractor's self-assessed reliability. node_type is one of the seven types above. scope is either PROJECT or TEAM, distinguishing knowledge specific to one codebase from knowledge that applies across projects.

3.3 Edges

Edges encode causal structure. triggered_pivot links a DEAD_END to the PIVOT that resulted. informed links a HYPOTHESIS to the INVESTIGATION it motivated. resolved links a DEAD_END to a downstream SOLUTION. These relationships transform isolated reasoning moments into debugging arcs, which is the unit of generalization the consolidation pipeline targets.

4 Two-Tier Extraction

CMM extracts reasoning DAGs from raw JSONL session transcripts via two complementary tiers.

4.1 Warm Tier: Heuristic Extraction

The warm tier runs in under one second per session with no API calls. It is triggered automatically by a stop hook wired into Claude Code's settings. Three independent strategies operate in parallel over the transcript:

Keyword classification compiles 3–7 regex patterns per node type. Confidence scales with match count: $0.35 + 0.1k$ for k matches, capped at 0.70.

Error-resolution pairing scans for an error message followed by a successful resolution within 8 messages. When found, it produces a paired DEAD_END + SOLUTION at confidence 0.60.

Explicit conclusion detection catches phrases like "I found that…" and "the issue was…" at confidence 0.65.

When two strategies independently flag the same message range, the higher-confidence node wins and receives a +0.05 corroboration boost (capped at 0.75). This produces immediate, free coverage at every session end. The warm tier is designed to never block: if any strategy fails, the others still produce output.

4.2 Cold Tier: Token-Budget LLM Extraction

The cold tier produces higher-precision nodes via an LLM but requires API calls. The naive design uses fixed message windows; we show this is suboptimal.

The fixed-window failure mode. A 200-message session windowed at 5 messages produces 60 LLM calls. A reasoning sequence spanning messages 8–16 (hypothesis at 8, investigation at 12, dead end at 14, pivot at 16) splits across windows. The model classifies each fragment in isolation and cannot infer the dead-end-to-pivot edge. The result is more API calls and lower-quality output.

Token-budget windowing. CMM measures each message's token cost using the Anthropic SDK's count_tokens method, then packs windows to a target fraction of the model's context window:

$$B = \lfloor M_{ctx} \cdot f \rfloor - T_{prompt} - T_{output} \quad (1)$$

With M_{ctx} = 200,000 tokens (Claude Sonnet 4.5), $f = 0.45$, T_{prompt} for the system prompt, and T_{output} = 4,000 reserved for the response, this yields $B \approx 85{,}000$ tokens per window. Adjacent windows overlap by 750 tokens to prevent splitting reasoning at boundaries. Each LLM call returns a list of 1–12 nodes, preserving extraction density across fewer calls.

Calibrating f. The 45% target is grounded in three published findings on long-context degradation. Chroma 2025 [5] shows monotonic performance decline across 18 frontier models with increasing input length. Liu et al. [7] report 30+% accuracy drops for mid-context information. Paulsen [10] demonstrates that effective context is drastically below advertised maximums. The 45% threshold balances seeing long reasoning arcs against avoiding the degradation zone. The parameter is exposed as CMM_CONTEXT_FILL_RATIO for empirical tuning.

4.3 Cross-Tier Upgrade

Warm and cold nodes coexist in the same store. When the cold tier processes a session previously covered by the warm tier, new high-confidence nodes pass through deduplication (Section 6). For pairs above cosine similarity threshold $\tau = 0.85$, the higher-confidence node wins, replacing the existing node if new, or being dropped if the existing has higher confidence. The store improves over time without manual reconciliation.

5 Consolidation: From Nodes to Patterns

After extraction and deduplication, the store contains hundreds of reasoning nodes. Consolidation transforms them into a structured set of architectural insights, frequency-ranked pitfalls with resolutions, and reusable diagnostic strategies. This consolidated structure is the substrate from which Skills are graduated.

5.1 Pipeline

Embedding. Each node summary is embedded as `"{node_type}: {summary}"` using OpenAI `text-embedding-3-small` (1536 dimensions). The type prefix improves clustering by separating semantically similar but typologically different nodes.

Clustering. Agglomerative clustering with cosine distance, threshold 0.4, average linkage. We use agglomerative rather than k-means because the number of natural groupings is unknown a priori and varies by project size.

Cluster classification. Each cluster of size ≥ 2 is classified by the LLM into one of three Skill components: architectural insight, pitfall, or diagnostic strategy. The LLM also assigns scope (`PROJECT` or `TEAM`) and produces type-specific fields: for pitfalls, severity and resolution strategy; for diagnostic strategies, trigger condition and ordered steps.

Pattern extraction. A separate pass extracts `key_patterns` (recurring approaches that work) and `anti_patterns` (approaches that consistently fail).

Assembly. The classified clusters and extracted patterns are written to ChromaDB and rendered as a markdown file, `cached_profile.md`, that any `SKILL.md`-compliant agent can load at session start.

5.2 Confidence-Prioritized Cluster Sampling

A cluster may contain dozens of nodes, but the LLM classification call samples a fixed number K for cost reasons. The choice of which K to send matters: it determines what the LLM sees as representative of the cluster.

CMM sorts each cluster by confidence descending before truncation, with $K = 10$:

```
sample = sorted(cluster,
    key=lambda n: n.confidence,
    reverse=True)[:K]
```

This ensures the LLM classifies each cluster from its highest-quality nodes. The system prompt directs the LLM to produce *transferable advice for future developers* rather than session narration:

> *You distill knowledge from AI coding sessions into advice for future developers. Every insight, pitfall, and strategy you produce must be specific enough that a developer who has never seen this codebase can act on it immediately. Describe what to do and why, not what happened.*

The cluster input format includes the `evidence` field alongside summaries, grounding the LLM's classification in concrete transcript content rather than paraphrase:

```
- [DEAD_END] provider factory missing branch
  Evidence: env var routes to wrong provider
```

`max_tokens` is set to 1500 to accommodate multi-step diagnostic strategies and full resolution descriptions.

The pseudocode appears in Algorithm 1.

Algorithm 1 Consolidation

Require: Nodes $\mathcal{N}$, threshold d, sample size K
$\mathcal{V} \leftarrow \text{embed}(\{(\text{type}_i, \text{summary}_i)\}_{i \in \mathcal{N}})$
$\mathcal{C} \leftarrow \text{AggClustering}(\mathcal{V}, d, \text{avg-linkage})$
$\mathcal{S} \leftarrow \emptyset$
for all cluster $C \in \mathcal{C}$ with $|C| \geq 2$ **do**
  $C_{sorted} \leftarrow \text{sort}(C, \text{by=confidence, desc})$
  $C_{sample} \leftarrow C_{sorted}[:K]$
  $\text{prompt} \leftarrow \text{build\_prompt}(C_{sample}, \text{w/ evidence})$
  $s \leftarrow \text{LLM}(\text{prompt}, \text{max\_tokens=1500})$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
end for
$\mathcal{P} \leftarrow \text{extract\_patterns}(\mathcal{N})$
return $\text{assemble\_skill}(\mathcal{S}, \mathcal{P})$

6 Storage and Retrieval

Storage. CMM uses ChromaDB as a persistent vector store. Each node is stored together with its summary text, 1536-dimensional embedding, and metadata (`project_id`, `node_type`, `confidence`, `scope`, `session_id`, `source_developer`, `retrieval_count`, `last_retrieved_at`). Storing text, vector, and metadata in a single collection makes retrieval a single query.

Deduplication. Before insertion, every new node is compared against existing nodes for the same project by cosine similarity. For pairs above $\tau = 0.85$, higher confidence wins. For pairs below, the new node is inserted. This produces both within-session and cross-tier deduplication uniformly.

Retrieval at session start. The cached profile is loaded directly via the `SKILL.md` mechanism: the agent's `CLAUDE.md` (or equivalent) instructs it to read `cached_profile.md` before any other action. No API call is required; the profile is plain markdown.

Retrieval mid-session. Four query commands are exposed as agent tools: `search-memory` for semantic search, `pitfalls` for the ranked pitfall list, `diagnose` for diagnostic-strategy matching against a problem description, and `cognitive-profile` for full-profile reload. Each invocation logs the query, retrieved nodes, and similarity scores to a SQLite evaluation database, and increments the `retrieval_count` of returned nodes. These logs and counts feed the retrieval-validation criterion in the graduation mechanism (Section 8).

7 Distributed Sync with Human Approval

A central design constraint is preventing hallucinated patterns from entering shared team knowledge. CMM operates in two modes: **LOCAL** (single developer, default) and **SHARED** (team-wide, opt-in). Shared mode connects local stores to a second ChromaDB instance and uses a three-stage workflow:

Push. A developer runs `cmm push`; new local nodes are uploaded to a *staging collection* marked `approved=False`. They are not visible to other developers.

Review. A reviewer runs `cmm review` and is presented with each pending node, including type, confidence, source session, and source developer. The reviewer can approve, reject, edit the summary, or reclassify scope between `PROJECT` and `TEAM`. Approved nodes are promoted to the main shared collection. All decisions are logged to a SQLite audit table with timestamp and reviewer identity.

Pull. Other developers run `cmm pull` and receive only approved nodes: project-scoped nodes for their current project, plus team-scoped nodes from any project.

The workflow mirrors version control (push, review, pull), making the semantics familiar. The human approval gate is the third graduation criterion (Section 8): a pattern must clear human review before it can be promoted to a Skill.

8 Skill Graduation

Memory and Skills sit at different points on a spectrum from fluid observation to stable procedure. Memory is liquid: it accumulates rapidly, evolves across sessions, and includes signal that has not yet been validated. A Skill is solid: a stable artifact that codifies established procedural knowledge for general consumption. The transition between the two should be empirical: patterns earn their way into Skills by surviving repeated observation, demonstrating retrieval value, and passing human ratification.

CMM's substrate makes this transition computable. We propose three graduation criteria:

1. Cross-session frequency ($f \geq f_{th}$). The pattern must recur. CMM tracks how many distinct sessions contributed to the cluster behind each consolidated insight, pitfall, or strategy. A pitfall encountered in 5 separate sessions is more likely to represent stable procedural knowledge than one encountered once. The threshold f_{th} is configurable; we set it to 3 distinct sessions as a default for project-scoped patterns and 2 for team-scoped patterns (since cross-project recurrence already implies generality).

2. Retrieval validation ($r \geq r_{th}$). The pattern must be useful. CMM's invocation logger records every retrieval: which memories were returned, with what similarity scores, in response to what queries. We compute the retrieval rate as the number of times a pattern was returned in the top-k for an actual agent query, divided by the number of agent sessions since the pattern entered the store. Patterns with high retrieval rates are validated by use. Patterns with zero retrievals after N sessions are demoted: they remain in memory but are excluded from the consolidated profile and from candidates for graduation.

3. Human ratification. The pattern must be approved by a reviewer. In shared mode, this is automatic: only approved nodes contribute to the shared profile, and only the shared profile feeds graduation. In local mode, ratification can be implicit (the developer keeps the pattern in their store) or explicit (a manual `cmm promote` command).

A pattern that meets all three criteria is graduated by exporting it as an entry in a stable `SKILL.md` file. The Skill is decoupled from the live memory store: it persists as a standalone artifact that can be versioned in source control, published to a Skill registry, or shared with collaborators outside the CMM workflow. This is the hand-off point between memory (liquid, local) and Skills (solid, portable).

The inverse path is equally important. CMM tracks each memory pattern's recency of last retrieval. Patterns that go unretrieved for an extended window have their confidence decayed (multiplicative factor 0.95 per session-week without retrieval, configurable). Patterns whose confidence drops below threshold are excluded from the consolidated profile. If a graduated Skill entry corresponds to a pattern that has decayed in memory, the Skill is flagged for review, since the Skill's underlying basis has weakened and a maintainer should re-validate it. This produces a Skill set that breathes with actual usage rather than accumulating indefinitely.

This mechanism directly addresses the 31% regression observed for self-generated Skills [2]. Self-generation produces Skills without empirical validation. CMM's graduation criteria require it: a pattern that has been observed across multiple sessions, retrieved successfully during real work, and ratified by a human is unlikely to fall into the regression class.

The retrieval tracking and confidence decay infrastructure required to evaluate the graduation criteria are implemented in this release. The `cmm graduate` command evaluates all consolidated profile items against the three criteria and exports eligible patterns as standalone `SKILL.md` artifacts. The longitudinal process of accumulating sufficient session frequency and retrieval signal for real patterns to reach graduation thresholds, and measuring the downstream effect of graduated Skills on agent performance, is the primary target of future empirical work (Section 11).

9 Evaluation

We evaluate CMM through a case study A/B comparison on a real bug in an enterprise codebase. The loaded artifact is `cached_profile.md` (the substrate's consolidated output) rather than a graduated `SKILL.md` in the sense of Section 8; this evaluation therefore tests the substrate's consolidation output rather than the full graduation pathway.

9.1 Setup

Codebase. We use a mid-sized enterprise open-source project: a multi-tenant authentication gateway. The codebase spans a Python web backend, a JavaScript frontend, multiple pluggable authentication providers, and deployment configurations for several cloud platforms. The bug we use was a known latent issue in the provider routing logic at the time of the evaluation.

Source data. Sessions from one developer over six months were ingested into CMM. The extraction pipeline produced reasoning DAGs that were consolidated into a profile containing architectural insights, pitfalls, and diagnostic strategies, all derived from real session evidence.

Task. A latent bug was selected as the comparison target: when the secondary authentication provider was configured via an environment variable, the management interface continued to display data from the default provider. The root cause was that the provider factory function read the environment variable as a module-level constant captured at import time and never re-evaluated it; the secondary-provider branch was missing entirely from the factory. The corresponding pitfall and resolution were present in the consolidated profile.

Conditions. The same task prompt was given to two agent configurations:

- **Baseline**: Claude Code with no `SKILL.md` loaded.
- **CMM-augmented**: Claude Code with `cached_profile.md` loaded at session start.

The CMM profile contained the relevant pitfall at high confidence, with a resolution describing the dynamic environment lookup pattern. Each condition was run once; variance across runs is not characterised in this evaluation. Section 11 discusses this and other limitations of the experimental design.

9.2 Results

Table 1 summarizes the comparison.

Table 1. Baseline vs CMM-augmented agent on identical bug-fix task.

Metric	Baseline	CMM	Δ
Assistant messages	119	46	−61%
Files modified	7	2	−71%
Session size (KB)	1,127	281	−75%
Root cause at message	48	37	−23%
Layers explored	7	2	−71%
Skill loaded at message	n/a	2	n/a

Trace analysis. The baseline began with broad codebase exploration: 8 `Bash` commands across the first ten messages, followed by twelve messages exploring frontend components, files with no causal relationship to the bug. The agent identified the root cause at message 48. The fix was distributed across 7 files, including frontend hooks, component templates, and tests for the touched components.

The CMM-augmented agent loaded the consolidated profile at message 2. At message 5 it opened the provider factory module, the file containing the bug. No frontend exploration occurred. At message 37 the agent articulated the root cause precisely: the provider factory read the environment variable from a module-level constant captured at import time, and the secondary-provider branch was missing. The fix was contained in 2 files: the provider factory module and one route handler.

Interpretation. The 61% reduction in messages and 71% reduction in files reflect more than speed; they reflect targeting. Without the loaded knowledge, the baseline hypothesized the bug could be in any of seven layers and explored each in turn. With the loaded knowledge, the augmented agent received the bug's location and resolution as initial context, eliminating six layers of unnecessary search. The Skill's resolution (read the environment variable dynamically inside the factory rather than capturing it at import time) directly informed the fix, which the augmented agent implemented exactly as described.

This is the regime in which Skills pay off: when stable procedural knowledge derived from real prior reasoning is loaded at session start, an agent's exploration becomes targeted rather than exhaustive.

10 Discussion

The CMM substrate gives Skills an empirical foundation. Three implications for the broader Skills ecosystem follow.

Skills as the artifact, memory as the substrate. Static Skills are valuable because they are stable, portable, and inspectable. But Skills authored without observational data are unmoored from real usage; the manual-curation gap and the self-generation regression both stem from this. CMM's graduation mechanism inverts the authoring direction: instead of writing a Skill and hoping it generalizes, CMM observes generalization (cross-session frequency, retrieval validation) and promotes only patterns that have already proven themselves. A Skill produced by graduation is a compact record of patterns the agent has used successfully, ratified by a human, and matched against real session queries.

The team as the unit of Skill authoring. CMM's distributed sync with human approval makes the team the natural Skill author. A pattern observed by one developer is staged, reviewed, approved, and propagated. Over time, the team's collective experience accumulates as a body of validated procedural knowledge that any team member's agent can load. This addresses the practical reality that procedural knowledge is rarely individual: the recurring pitfalls of a project are discovered by whoever happens to encounter them first, and the team benefits when that discovery is systematically captured and shared.

Skills with lifecycle. Static Skills have no built-in mechanism for staleness. A pitfall that was true six months ago may have been fixed by a library update; a diagnostic strategy that worked in version 2 may be obsolete in version 3. CMM's confidence-decay mechanism addresses this: graduated Skills whose underlying memory patterns have stopped recurring in recent sessions are flagged for review. The Skill set remains in step with the codebase rather than ossifying.

Composition with platform Skills. CMM-graduated Skills are standard `SKILL.md` files. They compose with hand-authored Skills, platform-provided Skills, and other community-contributed Skills through whatever loading mechanism the agent harness supports. CMM's contribution is at the authoring layer; the consumption layer is unchanged.

Generalization. CMM is currently specialized to coding agents, where the reasoning DAG taxonomy aligns naturally with debugging and feature work. The pipeline structure (extract reasoning, deduplicate, cluster, consolidate, graduate) generalizes to other agent domains by adapting the node taxonomy. The graduation criteria (frequency, retrieval, human ratification) are domain-independent and apply to any setting where Skills are loaded into agents at runtime.

Reproducibility and open release. The full implementation is released open source under Apache 2.0 with 216 tests, including the extraction pipeline, the consolidation algorithm, the distributed sync layer, the human review CLI, confidence decay, and the `cmm graduate` command. The case study used in our evaluation, including the source transcripts, the consolidated profile, and the comparison harness, is included in the release for reproducing the evaluation in this paper.

11 Limitations

Evaluation scope. The evaluation in Section 9 is a single-run case study on one bug in one codebase, drawn from one developer's prior session history. LLM agents are stochastic; a single observation per condition does not characterise variance, and the results should not be read as stable effect-size estimates. Multi-run replication and evaluation on additional bugs are the primary near-term extensions.

Conflation of substrate value and targeted-hint value. The comparison cannot distinguish two hypotheses: (a) CMM's extraction and consolidation pipeline produces knowledge that helps the agent, versus (b) loading any relevantly targeted prior knowledge helps. The consolidated profile contained the root cause and resolution at high confidence; a hand-written one-paragraph hint extracted from the same transcripts might produce similar improvement. Isolating CMM's contribution requires a baseline condition in which a human writes an equivalent hint without CMM's machinery; this is an experiment we treat as a priority for follow-up evaluation.

Graduation validated mechanically, not longitudinally. The `cmm graduate` command implements and enforces the three criteria. The longitudinal process of accumulating frequency and retrieval signal until patterns reach thresholds, then measuring whether graduated Skills improve downstream agent performance relative to the consolidated substrate, has not been empirically evaluated. Section 8 presents the mechanism; end-to-end empirical validation is future work.

Hyperparameter sensitivity. The thresholds used throughout CMM ($f = 0.45$, $\tau_{cluster} = 0.4$, $\tau_{dedup} = 0.85$, $f_{th} = 3$, decay rate = 0.95) are design choices calibrated against the literature or set as reasonable defaults. Sensitivity analysis across these parameters has not been performed and is left to future work.

Single-developer corpus. The 43 sessions in the case study were produced by one developer on one codebase. CMM's distributed sync and team-scoped pattern propagation are architectural features whose empirical validation in a multi-developer setting remains future work.

12 Conclusion

We presented CMM, a memory substrate for coding agents that captures the structure of agent reasoning, accumulates patterns across sessions and developers under a human approval gate, and provides the empirical foundation for graduating validated patterns into stable Skills. We showed that on a real bug in a mid-sized enterprise codebase, an agent loading CMM-derived knowledge at session start reduced its messages-to-solution by 61%, modified 71% fewer files, and reached the correct root cause 11 messages earlier than a baseline agent without access to the same knowledge.

The Skills ecosystem currently sits between two regimes: hand-curated Skills that work but do not scale, and self-generated Skills that scale but regress in a third of cases. Neither regime is acceptable as a long-term answer. We argue that an observational memory layer, one that watches real agent execution, captures the structure of what the agent reasons about, and graduates validated patterns into stable artifacts, is the missing piece. CMM is one realization of that argument, with concrete extraction, consolidation, distribution, and graduation mechanisms, and an open-source implementation that can serve as a starting point for follow-up work.

References

[1] Anonymous. 2026. A Security Audit of 3,984 Community-Contributed Agent Skills. *arXiv preprint* (2026).
[2] Anonymous. 2026. SkillsBench: Benchmarking the Impact of Procedural Knowledge on LLM Agent Performance. In *Proceedings of the First Workshop on Agent Skills (Agent Skills '26).* To appear.
[3] Anthropic. 2026. Agent Skills. https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview.
[4] Anthropic. 2026. Claude Code: Auto Memory and Auto Dream. Product release notes.
[5] Chroma Research. 2025. Context Rot: A Study of Context Length Performance Degradation Across 18 Frontier Models. Technical report.
[6] Cursor Inc. 2024. Cursor: AI-Native Code Editor. https://cursor.sh.
[7] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. In *Transactions of the Association for Computational Linguistics.*
[8] Mem0 Inc. 2025. Mem0: The Memory Layer for AI Agents. https://mem0.ai.
[9] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2023. MemGPT: Towards

LLMs as Operating Systems. In *arXiv preprint arXiv:2310.08560*.

[10] Anonymous Paulsen. 2025. Maximum Effective Context Window: Empirical Bounds on Frontier LLM Recall. *arXiv preprint* (2025).

[11] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

[12] Topoteretes. 2024. Cognee: Memory for AI Agents. https://github.com/topoteretes/cognee.

[13] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. ExpeL: LLM Agents Are Experiential Learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*.