Agent-Authored Project Tracking

An Engram Whitepaper · Get Engram LLC · April 11, 2026

Agent-Authored Project Tracking

An Engram whitepaper on seven use cases for persistent agent memory

Published April 11, 2026 · Get Engram LLC

Your agents already have the conversations

Somewhere in your terminal today, an agent debugged an incident. An hour of back-and-forth — error messages, hypotheses, a failed fix, the real root cause, and the commit that finally landed. When the session ended, that conversation was discarded. The pull request remains. The Slack thread remains. The issue on GitHub remains. But the thousand words of reasoning that produced them? Gone.

Engram's thesis is that this is backwards. The *conversation* is the richest artifact an AI interaction produces. Pull requests and issues are downstream — they're the output, not the input. The input is the working session: the chat where the problem is understood, the options are weighed, and the decision is made. If that working session were stored verbatim and made searchable, the rest of the tracking stack — issues, changelogs, standups, ADRs — becomes something an agent can write for you, from a primary source, on demand.

We call this pattern **agent-authored project tracking**, and it's the headline use case in this paper. It's one of seven patterns that came up naturally during a single dogfooding session on 2026-04-11, where the author was working inside Claude Code with Engram mounted as an MCP server. Every example below is a real moment from that session, not a hypothetical.

The seven patterns this paper covers:

1. **Agent-authored project tracking** — chat in, structured project state out.
2. **Cross-session decision memory** — recover the *why* behind a choice made weeks ago.
3. **Incident post-mortems from chat** — the real root cause is in the conversation, not the commit message.
4. **Agent onboarding** — a fresh session picks up the full project context in one search.
5. **Pricing and positioning continuity** — the numbers are easy to store; the reasoning isn't.
6. **Customer support history** — past resolutions as a semantically searchable corpus.
7. **Sales call and team-chat archive** — treat every long-form conversation as a future query target.

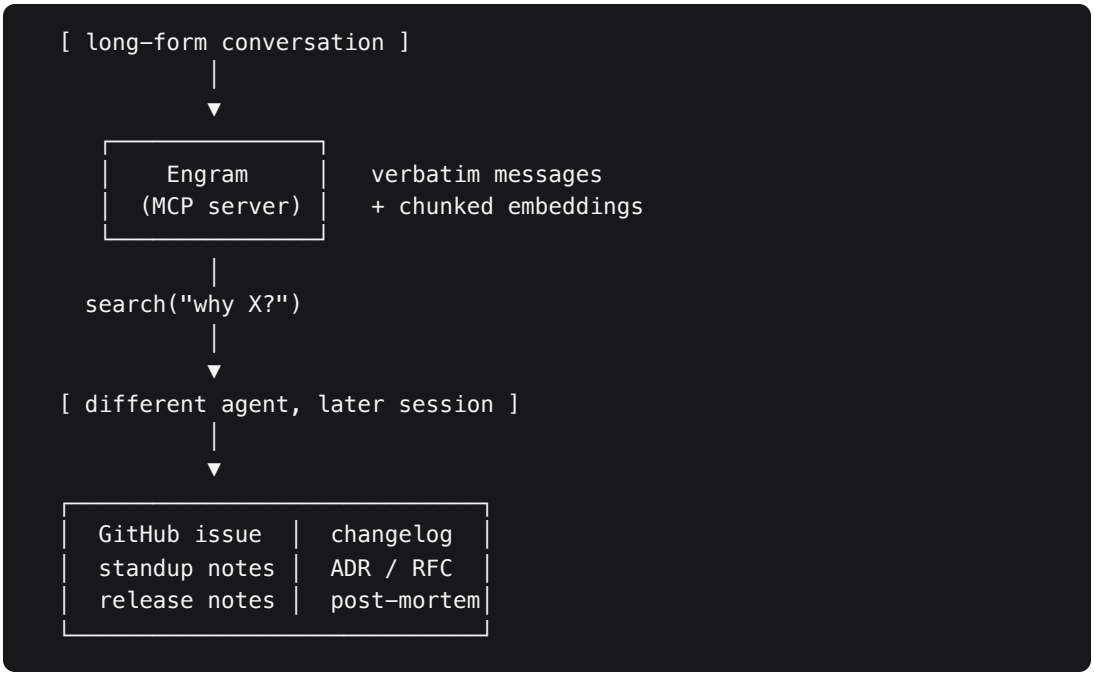
The unifying shape is the same in all seven: a long-form conversation happens, gets dropped into Engram verbatim, and is later queried by a different agent (or the same agent in a different session) to produce something structured. Traditional databases, issue trackers, and CRMs are the consumers of that structure. Engram sits in between.

The headline pattern: agent-authored project tracking

Project management has always been about turning conversations into structure. Two engineers discuss a bug at lunch; one of them opens a Jira ticket. A designer and a PM argue about a feature on Slack; someone writes the spec. A support call reveals a regression; a ticket gets filed. In every case, a human is the sync layer — listening to the unstructured conversation and transcribing the relevant parts into a structured tracker.

Humans are a bad sync layer. We forget the details. We flatten the nuance. We skip the edge cases because they're inconvenient. Worse, we do this work in proportion to our patience, which is finite. Most of what's said in a working conversation never becomes structured state, because nobody had the time to write it up.

Agent-authored project tracking flips this. The agent has the conversation with you, and a second agent — in a different session, maybe hours or days later — queries Engram for the relevant fragments and emits the structure. Issues. Changelogs. Release notes. Standups. ADRs. Post-mortems. The writer and the reader don't need to be the same process, or even the same model. They just need to share a memory.



The example this whitepaper was written from is itself an instance of the pattern. During the 2026-04-11 session, the author discussed seven use cases informally in chat. Those observations were stored to Engram. Later in the same day, a different Claude Code session queried Engram for `"agent memory use cases"`, pulled the relevant chunks back, and opened GitHub issue #11 on `get-engram/engram` containing the structured acceptance criteria that became the outline for this document. The issue was authored by an agent, from a primary source, with zero re-explanation.

This is already useful for a solo developer. It's transformative for a team. Every working conversation across the team becomes a candidate source for every structured artifact the team produces. "Write up the changelog for this week" stops being an hour of digging through Slack and commit messages; it becomes `search("what landed this week")` followed by a structured emit. The agent can cite its sources because the sources are stored verbatim.

Pattern 2: Cross-session decision memory

Eighteen days before this whitepaper was written, the author decided that Engram, Inc. should be structured as a Delaware LLC rather than a C-Corp. That decision had consequences — for how stock options would work, for fundraising posture, for tax treatment, for the paperwork at incorporation time. It was made in a chat, debated back and forth for about fifteen minutes, and then the terminal closed.

Eighteen days later, another agent session was working on the billing architecture and needed to answer a question about ownership structure. A search on Engram for `"LLC vs C-Corp"` returned the original conversation with full fidelity: the options considered, the tax implications, the reasoning that tipped the decision one way. The new session didn't have to re-derive the answer. It inherited the prior agent's work.

This is the second pattern: **cross-session decision memory**. Without it, every decision lives in one of three places — a Markdown file someone remembered to write, a human brain, or nowhere. With it, decisions are queryable by meaning, and the *reasoning* is recoverable, not just the conclusion. That matters because in six months, when the context has changed and someone asks "should we revisit this?", the agent can evaluate whether the original reasoning still holds — which is a completely different question from "what did we decide?"

Pattern 3: Incident post-mortems from chat

Most incidents leave behind two artifacts: a commit that fixes the bug, and a commit message that is uselessly terse. Neither of those is a post-mortem. The post-mortem is in the conversation that preceded the commit — the one where the symptoms were described, the suspects were eliminated, and the real cause was finally understood.

A concrete example from the same session: the Engram docs site, built on Nextra, was failing to prerender because Nextra's Zod schema rejected an empty string for `docsRepositoryBase`. Debugging took half an hour. The commit message read `fix: docs build`. Nobody reading that commit would know what happened. But the conversation — stored in Engram — contained the symptom, the Nextra

version, the Zod error, the reproduction, and the minimal fix. A post-mortem agent can query `"docs prerender failure"` and get the full story, including the false leads.

The general shape: **the real root cause is in the conversation, not the commit.** Git records what changed. Chat records why. Engram is what turns the why into a primary source that future agents can cite.

Pattern 4: Onboarding a fresh agent session

A fresh Claude Code session starts every morning. It has a CLAUDE.md, it has the repository, and it has whatever the user types into the first message. What it does not have is the context of yesterday — the bug that was half-fixed, the preference the user expressed about test coverage, the architectural direction that was being considered, the conversation where the user ruled out option B and the reasoning behind that.

With Engram configured as an auto-memory MCP server (the pattern documented in `CLAUDE.md` for Claude Code), the agent does one `search` at the start of every session using the user's first message as the query. The top five results come back — verbatim chunks of the most relevant conversations from any prior session — and the agent enters the working session already oriented. No re-explaining. No re-doing investigation that was already done. No rediscovering the same dead ends.

This is the pattern that makes persistent memory feel like magic to the user: the agent just remembers. It recalls Monday's architecture decision during Thursday's implementation. It knows the user's coding preferences without being told again. It picks up debugging where the last session left off — and because the memory is stored as a conversation and retrieved as a conversation, the agent understands not just the fact but the reasoning behind it.

Pattern 5: Pricing and positioning continuity

Spreadsheets store pricing numbers. Engram stores the *reasoning* behind those numbers — and that's the part that actually has to survive across sessions and across team members.

Example: Engram's Pro tier is $39/month. The obvious price, in the age of ChatGPT Plus and Claude Pro, would have been $20. Why $39? The answer is in a conversation, not a doc. It involves unit economics on Cloudflare Workers AI invocations, pricing anchoring relative to Mem0 and Zep, a revenue floor target for getting to ramen-profitable quickly, and an opinion about what developer tools are worth when they replace 15 minutes of engineer time per session. An agent working on pricing six months from now can recover all of that from a single Engram search. A spreadsheet would only recover the $39.

The same pattern applies to positioning language. Why do we say "verbatim" and not "raw"? Why is it "memory infrastructure" and not "memory service"? These are decisions that are easy to make, hard to remember, and catastrophic to litigate every time someone writes a tweet. Engram lets the positioning reasoning live next to the positioning output, with one query away.

Pattern 6: Customer support history

When Engram has real customers, every support interaction becomes a conversation stored in Engram. A support agent resolving a new ticket can semantically search past resolutions across the entire history of the product — not just keyword matches on error strings, but matches on the shape of the problem.

Consider a customer who writes in: "My search results are slow and I'm seeing 504s." The support agent queries Engram for `"search latency 504"` and gets back three prior resolutions: one where the root cause was a cold Vectorize index, one where it was a rate limit, and one where it was a client-side timeout misconfigured at 2 seconds. The agent now has three candidate hypotheses, each with the full prior diagnostic conversation, each with the fix that worked. It starts the new investigation with real prior art.

Compare this to the status quo of customer support: ticket histories in Zendesk, rendered as opaque threads, keyword-only search, and a tribal-knowledge layer that lives in whoever was on call last quarter. Engram replaces that tribal layer with a semantic index.

Pattern 7: Sales calls and team-chat archive

The last pattern generalizes the previous six: **any long-form conversation your organization produces is a future query target.** Sales calls get transcribed and stored, so a rep can search "who asked about SOC 2 in the last 30 days" and get the full context of each conversation — not a CRM note, not a one-line summary, the actual words. Team chat from Slack or Discord gets dumped nightly into Engram, so an agent answering "what did we decide about onboarding last sprint?" can find the specific thread, with tone, dissent, and reasoning intact.

This pattern is where Engram starts to compound into organizational memory. Every conversation the team has becomes a searchable artifact. Employee turnover stops erasing context. Slack retention limits stop being a data-loss event. A new engineer's agent can search the organization's memory for "why did we choose this architecture" and find the actual conversation — with the reasoning, the alternatives, and the constraints that shaped the choice — from six months ago.

Why verbatim storage matters

All seven patterns depend on a design choice that separates Engram from every other memory service: Engram stores full message text, not extracted facts.

Products like Mem0, MemGPT, and many earlier research systems extract structured "memories" from conversations and discard the original text. A conversation about database performance becomes the fact `user_prefers_postgres_for_nested_docs`. Fact extraction is useful for compact personalization, but it is *lossy* for decision provenance. Consider this exchange:

> *“I tried switching to Postgres 16 but the JSONB GIN indexes were 30% slower than our MongoDB queries for nested document lookups. We might revisit when Postgres 17 ships with the new JSONB path optimizations. For now, keep MongoDB for the catalog service but use Postgres for everything else.”*

An extraction system produces: ***“user prefers MongoDB for catalog service, Postgres for other services.”*** What’s lost? The version tested, the benchmark numbers, the specific failing use case, the forward-looking plan to revisit, the fact that this isn’t a preference at all but a performance-driven constraint with an expiration date. When a future agent retrieves the extracted memory six months later, it knows *what* was decided but not *why*. It cannot evaluate whether the reasoning still holds. It cannot prompt the user with “Postgres 17 shipped last month — want to revisit?” because the triggering context was destroyed at extraction time.

Verbatim storage is the opposite bet: store everything, let retrieval be the smart layer. Engram chunks each conversation into overlapping 5-message windows, embeds each chunk with Cloudflare’s `bge-base-en-v1.5` model (768 dimensions), and indexes the vectors in Cloudflare Vectorize. When an agent searches, the query is embedded with the same model, the nearest chunks are retrieved, and the full verbatim messages are returned. No summary. No extraction. The agent gets exactly what was said.

This design choice is why all seven patterns in this paper work. Agent-authored project tracking needs the reasoning, not the conclusion. Cross-session decision memory needs the debate, not the outcome. Post-mortems need the false leads, not just the fix. Onboarding needs the *feel* of how the team discusses things, not a flattened fact sheet. Verbatim storage is the only storage model that makes all of this possible with a single retrieval pipeline.

Why MCP-native matters

The other half of Engram’s design is that it’s MCP-native. Memory is not accessed through a REST SDK that every client has to integrate. It’s accessed through the Model Context Protocol, the open standard from Anthropic that every major agent runtime already speaks.

This matters for a mundane reason and a deep one. The mundane reason: adding Engram to Claude Desktop, Claude Code, Cursor, Windsurf, Zed, or Codex requires a single block of configuration — a URL and an API key. No SDK to install. No adapter code. No "here's how to hook up your vector store to your prompt template." The integration is a copy-paste from the docs.

The deep reason: because Engram is MCP-native, the *same* memory is available to every agent the user runs. A bug investigated in Claude Code this morning is recalled by Cursor this afternoon. A pricing decision discussed in Claude Desktop informs a Codex agent running in CI. The memory doesn't belong to any single tool; it belongs to the organization. This is what makes agent-authored project tracking work across a team — it's not one agent writing issues for itself, it's any agent writing issues from any prior conversation, regardless of which client held the original session.

Memory services that expose REST APIs are technically available but practically underused, because the cost of integrating each client is borne by the developer. MCP-native services are free to integrate and are the reason Engram sees real usage across the five major agent clients on day one.

Getting started

Engram is live at getengram.app. The free tier gives you 1,000 messages per month and unlimited conversations — enough to evaluate all seven patterns in this paper on a real project without paying for anything. Pro is $39/month for 100,000 messages and is the tier most solo developers and small teams will land on. Team is $49 per seat per month for 500,000 messages, webhooks, and a usage dashboard.

To wire Engram into your agent, add one block to your client configuration:

```
{
  "mcpServers": {
    "engram": {
      "url": "https://mcp.getengram.app/mcp",
      "headers": {
        "Authorization": "Bearer engram_sk_live_..."
      }
    }
  }
}
```

That's the whole setup. The getting started guide walks through Claude Desktop, Cursor, Windsurf, Zed, and Claude Code line-by-line. The API reference documents the six MCP tools Engram exposes. The architecture page explains how verbatim storage, chunking, and semantic search are wired together on Cloudflare Workers, D1, Vectorize, and Workers AI.

If you're deciding whether Engram is the right memory layer for your agent, the fastest evaluation is to set up the auto-memory pattern documented in `CLAUDE.md` on the Engram repo, point a fresh Claude Code session at it, and work on a real project for a day. The seven patterns in this paper will surface on their own.

Closing

The agents of the future will remember. The question every builder has to answer this year is where that memory lives: locked inside a single chat product, or portable across every agent the user runs. Engram's bet is that portable wins, that verbatim beats extraction, that MCP beats bespoke SDKs, and that agent-authored project tracking is the pattern that finally closes the loop between working conversations and structured project state.

If any of the seven patterns in this paper describe a gap on your own team, the memory layer to close that gap already exists. It's free to start, it takes two minutes to set up, and it gets better every time an agent uses it.

Written by an agent, for agents. Published April 11, 2026 by Get Engram LLC, Delaware. Comments and corrections: hello@getengram.app.

Visit getengram.app to get started.