AI Disclosures Project

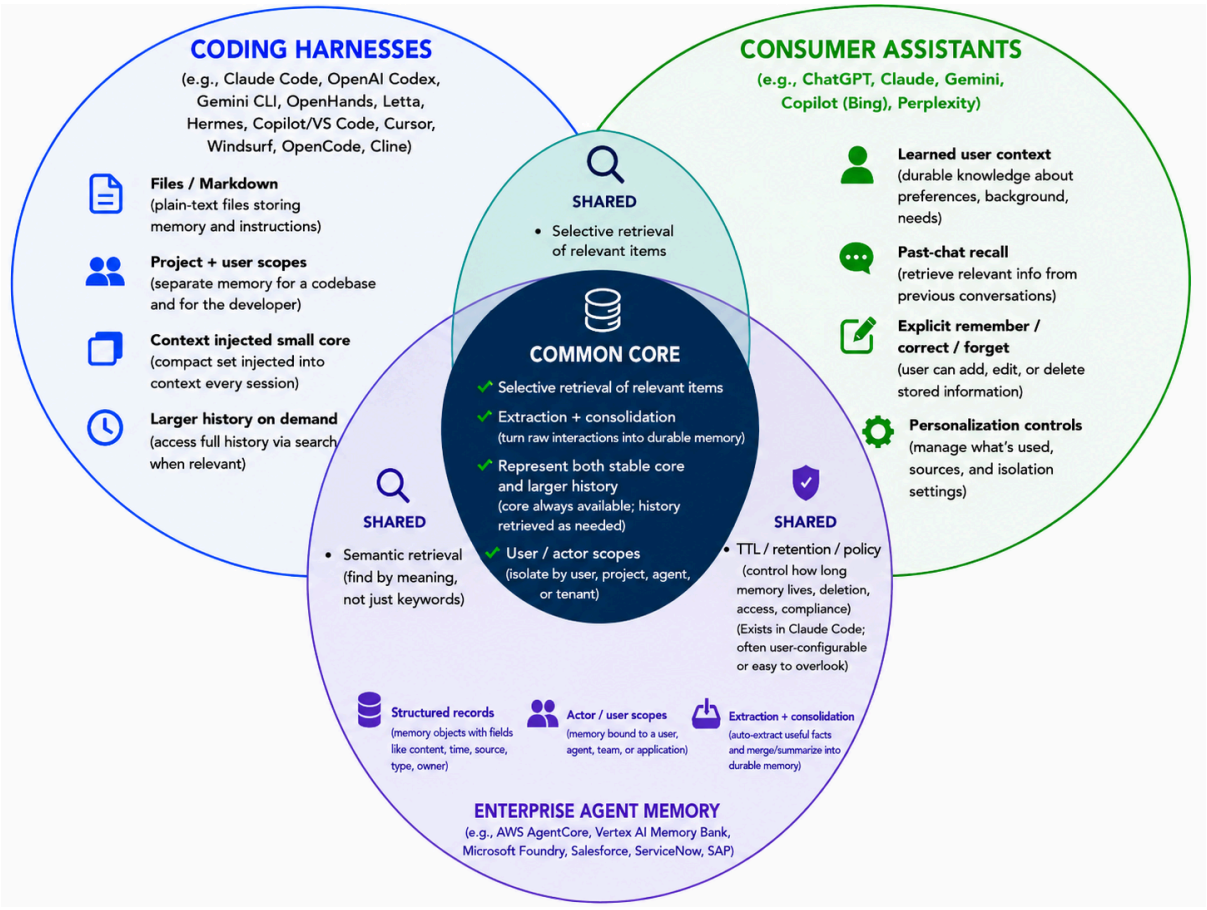
An Open Protocol for Agentic Memory

Scoping Note: Shared features of agentic memory systems

Last Updated: August 2026

Summary

Persistent memory is becoming a necessary layer of agent systems, but there is still no common way to represent or move it across products. This brief surveys current practice across coding harnesses, consumer assistants, and enterprise memory systems to identify the smallest set of shared primitives that could support an open memory protocol. In addition to the use of a common storage syntax (.md files), there is some convergence around memory persistence, scope and ownership, lifecycle operations, retrieval, and controls.[1]



---

[1] Strauss, I., & Rosenblat, Y. S. (2026). *Scoping Note: Shared Features of Agentic Memory Systems*. AI Disclosures Project. August 27, 2026. Doi: 10.5281/zenodo.22234513.

1. Why memory is becoming a protocol question

Memory is part of the context layer that allows AI agents to personalize conversations and carry useful knowledge across tasks and sessions. Today that memory is usually bundled into a particular product, coding harness, or provider-hosted store. Consumer providers can export raw chat histories, but this requires waiting 24 hours and is not the derived memory objects that shape model behavior. Coding and enterprise systems expose richer memory, but in incompatible formats. This creates switching costs for users that will grow with time and duplicated engineering work for developers or enterprises using multiple different agentic products concurrently.

An open protocol can provide a shared language for memory so that it can become unbundled from any one product by providing a common schema (data structure / fields) and associated semantics (what the fields mean) to transfer across products. This does not necessarily mean standardizing how memory is stored internally. Companies can still innovate while keeping the underlying data portable. A thin interoperability protocol layer might cover common memory records, their scope and ownership, provenance, lifecycle operations, and exchange semantics. With this goal in mind, this note identifies shared memory features across production AI systems: focusing on what they already do rather than designing a new memory architecture from scratch. Memory, as a technology, is still evolving and so any attempt at hardening inflexible standards around it is unlikely to succeed at this stage.

2. Common practices across current memory systems include

1. Persistent memory across sessions**:** Memory survives beyond the current conversation or run, so information can be carried forward into later sessions. Claude Code, OpenHands, Hermes, VS Code/Copilot, Deep Agents, ChatGPT, Gemini, AWS AgentCore, Vertex Memory Bank, and Microsoft Foundry all implement some form of persistent memory.

2. Scope and ownership: Memories are tied to a particular owner or context, such as a user, project, repository, agent, team, or organization. The terminology differs, but the basic idea is widespread: a system needs to know *whose memory it is and where it applies*. Scope and ownership are likely to be fields in a common memory schema.

3. Selective retrieval: Systems increasingly retrieve only the memories relevant to the current task rather than loading the entire memory store. The retrieval method varies by product, but selective access is common across coding harnesses, consumer assistants, and enterprise memory systems.

4. Extraction and consolidation: Systems increasingly turn raw conversations or activity into more durable memory by extracting useful information and then merging, summarizing, updating, or replacing older entries. The details differ, but this basic process now appears across all three groups.

5. Common memory operations (a memory lifecycle): Memory is generally something that can be created, recalled, changed, and removed. A minimal open protocol should therefore support basic operations such as *create* (remember), *read* (recall), *update* (correct, consolidate, or supersede), and *delete* (forget), while allowing systems to apply their own privacy, access, retention, and provenance rules.

A *small always-loaded core plus a larger history* is a strong pattern in coding harnesses and some assistants, but it is not necessary for enterprise agent memory and should not be treated as a universal feature. Similarly, *semantic retrieval* (finding memories by meaning rather than exact words) is especially common across consumer assistants and enterprise memory systems, while the precise retrieval method should remain implementation-specific.

2.1 File-backed memory **that the user can inspect but is not yet portable**. In brief:

- Claude Code uses `MEMORY.md` + topic files;
- OpenHands uses `MEMORY.md` + dated logs;
- Hermes has tightly bounded `MEMORY.md` and `USER.md`;
- Gemini CLI edits hierarchical Markdown;
- Letta increasingly exposes structured memory blocks/MemFS.

These are conceptually similar but not mutually portable formats. Not all of them have version control. Letta is Git-backed, while Claude Code auto-memory and Windsurf/Cascade memories are machine-local.

3. Patterns by system type

Coding harnesses

Coding harnesses increasingly use Markdown files to store memories. These files can be stored at the project level, scoped to the currently opened project, or at the user level where they are available across folders. Many memory systems inject a small compact summary of what the model knows into the model's context (through things like AGENTS.md), while placing the rest in deferred and searchable memory (larger files that are not loaded automatically but can be retrieved when relevant). Key points:

- Files / Markdown
- project + user scopes
- small always-loaded core
- larger deferred/searchable memory

Consumer assistants

Consumer assistants increasingly rely on both explicitly *extracted memories* (memories set by the agent or inferred from chat transcripts) and on access to *raw transcripts*, allowing the model to search or read chat history They often provide explicit remember, correct, and forget controls (ways for the user to add, revise, or delete stored information), alongside broader personalization controls (settings that determine whether memory is used, what

sources it can draw from, and sometimes which conversations or projects are isolated). Key points:

- learned user context
- past-chat recall
- explicit remember/correct/forget
- personalization controls

Enterprise agent memory

Enterprise systems are more varied but tend to store memory as structured records (individual memory objects with defined fields such as content, timestamp, source, owner, or type), assigned to actor or user scopes (boundaries specifying which person, agent, team, or application the memory belongs to). They commonly use extraction and consolidation (automatically identifying useful facts from interactions and merging or summarizing them into durable memory), followed by semantic retrieval (finding memories based on meaning rather than exact keyword matches). They also put more emphasis on TTL, retention, and policy (rules governing how long memory exists, when it is deleted or archived, and who is permitted to store or access it). Key points:

- structured records
- actor/user scopes
- extraction + consolidation
- semantic retrieval
- TTL / retention / policy

4. What is converging in agentic memory systems…and what is not

Background consolidation / "dreaming". The strongest new cross-vendor convergence. OpenAI, Anthropic managed agents, and Letta independently run background passes over accumulated history or memory to synthesize, deduplicate, or reorganize durable memory. The implementations differ, and there is no shared schema for recording what was merged, superseded, or invalidated.

Progressive disclosure. Load only the information that is needed into memory. Many systems keep a small core of important memory available at all times, while leaving the rest stored separately and pulling it in only when relevant. This is common in coding agents, but is not universal.

Skills / SKILL.md. Skills store reusable instructions for how an agent should perform a task, rather than facts the agent has learned or remembered. They can function like procedural memory, but they are better treated as a separate layer from persistent memory. Skills are not created and changed through regular use like memory.

Human approval of memory writes. Some systems require review before saving (for example Gemini CLI experimental auto-memory and Cursor); others write autonomously (for

example Claude Code, Hermes, and Letta). This is a product choice, not a protocol commonality.

How retrieval works internally. Systems differ in how they decide which memories to bring back: some search by meaning, some favor recent memories, some expose memories as files on a file system that can be read at the model's discretion, some limit how much memory can enter the prompt, and some use their own ranking rules. Those choices affect performance, but they do not need to be standardized for memories to be portable.

MCP memory access. MCP is a transport/interface layer: it defines how an agent can call tools to search, read, write, update, or delete memory. It does not define the memory itself: its format, schema, scope, or lifecycle. There is not yet a widely adopted MCP standard for memory-specific pieces.

5. Where the protocol gap is

Cross-provider interchange is still primitive. Claude's current import flow tells users to ask another assistant to export its memories as text or Markdown and paste the result into Claude. In practice, copy-and-paste is the closest thing to a cross-provider interchange format.

The protocol opportunity is therefore not to standardize storage engines. A minimal spec should define: (1) a portable memory record: content, scope/owner, provenance/source, timestamps, and lifecycle operations; (2) a lineage/consolidation record containing what was merged, superseded, or invalidated, when, and why. Markdown, vector stores, graphs, embeddings, and ranking algorithms can remain implementation-specific.

Appendix: Existing memory systems

Verified against first-party documentation by ChatGPT and human reviewers (current as of August 2026).

- **Claude Code**: Separates user-authored `CLAUDE.md` instructions from **auto memory** (notes Claude writes itself from preferences, corrections, project context, and references). Auto memory is **on by default** and stored per repository under `~/.claude/projects/<project>/memory/`, with a `MEMORY.md` index plus individual topic files. Only the first 200 lines or 25KB of `MEMORY.md` are loaded at session start; topic files remain deferred and are read on demand. Auto memory is machine-local and shared across worktrees of the same repository. Claude Code also supports Agent Skills, which load on demand. (Claude Platform Docs)
  Auto-memory notes also expose a useful primitive schema: a type field in YAML frontmatter (user, feedback, project, or reference); when Claude writes a memory file that already has frontmatter, Claude Code also records a modified timestamp in ISO 8601 format (v2.1.214+).

- **OpenAI Codex**: Uses learned memory derived from prior sessions through a **two-stage extraction and consolidation pipeline**. Its memory workspace, normally under `~/.codex/memories/`, includes `MEMORY.md`, a compact `memory_summary.md`, per-session `rollout_summaries/`, and optional learned `skills/`. `memory_summary.md` serves as compact prompt-loaded context; Codex can then search `MEMORY.md` and selectively open relevant rollout summaries or skills. Project instructions use `AGENTS.md`, while skills use `SKILL.md` entrypoints. (GitHub)

- **Gemini CLI**: Uses hierarchical `GEMINI.md` context files at global and workspace levels, supports imports, and discovers nested `GEMINI.md` files **just in time** when tools access relevant files or directories. It also has an **experimental Auto Memory** system that mines past sessions for durable facts and reusable skills, but proposed memory changes and `SKILL.md` files are held for **user review and approval** rather than applied automatically. Gemini CLI also supports Agent Skills and MCP. (Gemini CLI)

- **OpenHands**: Provides an **opt-in, two-tier persistent memory system** (off by default) with user memory at `~/.openhands/memory/` and project memory at `.openhands/memory/`. Each tier contains a compact `MEMORY.md` index plus dated daily logs. Only the two `MEMORY.md` indexes are injected automatically, with a combined budget of roughly 6,000 characters; daily logs remain deferred and are opened on demand. OpenHands keeps `AGENTS.md` for explicit instructions and separately supports Agent Skills. (OpenHands Docs)

- **Letta**: Uses **MemFS**, a Git-backed memory filesystem that is part of each agent's persistent state. Memories appear as ordinary Markdown files: files under `system/` are loaded into the system prompt on every turn, while files elsewhere remain deferred. The filesystem tree is always visible to the agent, allowing it to selectively open relevant files. MemFS does **not** use semantic/vector search by default; that can be added through an optional search extension. Letta also supports Agent Skills. (Letta Docs)

- **Deep Agents / Deep Agents Code**: LangChain's Deep Agents provides first-class, filesystem-backed long-term memory with configurable storage backends and explicit **user, agent, and organization scopes**. Memory files are normally loaded into the prompt at conversation start and can be updated by the agent; past conversations provide episodic memory through persisted threads, while skills provide procedural memory and load on demand. Deep Agents explicitly distinguishes **semantic memory** (facts), **episodic memory** (past experiences), and **procedural memory** (instructions and skills), and optionally supports background consolidation of conversations into durable memory. (Docs by LangChain)

- **OpenClaw**: Uses a transparent Markdown workspace with `USER.md` (stable user preferences/profile), `MEMORY.md` (compact durable facts and decisions), and dated files under `memory/` (detailed working and episodic notes); it also supports an optional `DREAMS.md` for human-reviewable dreaming output. `USER.md` and

`MEMORY.md` form the compact startup layer, while older daily notes are indexed and available through `memory_search` and `memory_get`. OpenClaw's **dreaming** process can distill useful information from episodic notes into durable `MEMORY.md`. (OpenClaw)

- **Hermes Agent**: Uses two bounded native memory files: `MEMORY.md` (agent notes, environment facts, conventions, and lessons) and `USER.md` (user preferences and profile), stored under `~/.hermes/memories/`. Both are injected as a frozen snapshot at session start and have strict character limits; the agent can add, replace, remove, and consolidate entries. Larger historical context remains available separately through full-text `session_search` over past conversations. `SOUL.md` is a separate personality/context mechanism, **not part of Hermes's native persistent-memory store**. (Hermes Agent)

- **GitHub Copilot / VS Code**: VS Code now provides a built-in **memory tool** with user, repository, and session scopes exposed through the virtual `/memories/` namespace. User memory persists across projects and automatically injects its first 200 lines; repository memory persists within a workspace; session memory disappears when the chat ends. Repository memory can optionally be backed by **Copilot Memory**, GitHub's separate hosted repository-memory system, which shares learned insights across Copilot agents and automatically expires them after 28 days. VS Code also supports `AGENTS.md` and Agent Skills. (Visual Studio Code)

- **Cursor**: Supports **Memories**, automatically generated rules extracted from conversations and scoped to an individual user within a project/repository. Background-generated memories require user approval before being saved and can be viewed or deleted through Cursor's settings. Cursor Automations additionally have a memory tool that allows cloud agents to learn across repeated runs, with memory files editable or deletable through the UI. This memory is product-managed rather than based on a standardized portable filesystem convention. Cursor separately supports `AGENTS.md`, project rules, user rules, and skills. (Cursor Documentation)

- **Windsurf / Cascade**: The legacy **Cascade** agent uses automatically generated, workspace-scoped memories stored locally under `~/.codeium/windsurf/memories/`; relevant memories are retrieved automatically rather than loaded wholesale. However, this feature now applies **only to the legacy Cascade agent**: the newer Devin Local agent does not persist Cascade memories, and the documentation recommends migrating durable knowledge into Skills, Rules, or `AGENTS.md`. (Devin Docs)

- **OpenCode**: Does not currently expose a distinct first-party **learned-memory system** comparable to Claude Code auto memory or Cursor Memories. Instead, OpenCode V2 emphasizes persistent instructional context through `AGENTS.md`, skills, references, MCP, and session context. It loads global and ancestor `AGENTS.md` files initially and discovers nested `AGENTS.md` files when the agent accesses deeper files or directories. This is best classified as **persistent context/instructions rather than learned memory**. (OpenCode)

- **Cline**: Its **Memory Bank** is explicitly a Markdown documentation methodology rather than an autonomous first-party learned-memory subsystem. A typical bank contains `projectbrief.md`, `productContext.md`, `activeContext.md`, `systemPatterns.md`, `techContext.md`, and `progress.md`. The recommended Cline Rule instructs the agent to read **all core Memory Bank files at the start of every task**, so this approach favors eager loading of a structured project state rather than progressive retrieval. (Cline)