08 — Memory & State

🕒 state of the art 2026-07 · revised 2026-07-26 · 📖 ~12 min read

Learning objectives

By the end of this chapter, you should be able to:

1. **Distinguish** the problem's three layers — session state, long-term memory and workspace state — and the requirement specific to each;
2. **Explain** why memory **is not** RAG (Retrieval-Augmented Generation) (memory = retrieval + write path + state management) and why versionable markdown beat vector databases in the code domain;
3. **Derive** a recall policy from the recency × importance × relevance formula, and a forgetting policy from usage;
4. **Evaluate** the impact of **reversibility** (workspace checkpointing) on the permissions risk calculus;
5. **Implement** harness-zero's session persistence (SQLite adapter + `/resume`) in step 4.

The problem

The model forgets everything between calls; the harness remembers for it. "Memory and state" covers three layers with different requirements:

1. **Session state** — the conversation itself: messages, tool calls, metadata. It must survive restarts and allow resuming (`resume`), branching and reverting.
2. **Long-term memory** — facts that cross sessions: user preferences, project decisions, learnings. It must be **selectable** (not everything enters every context) and **updatable** (facts change).
3. **Workspace state** — what the agent *did* to the files. It must be **reversible**: undoing an agent's changes is as important as making them.

The thesis unifying the three: the context window is volatile, expensive memory; everything that needs to last lives **outside** it, and the harness decides what to bring back and when.

Scientific foundations

Agent memory has a mature literature — and it provides the exact vocabulary for what the harnesses do in practice.

- **The window as RAM** — MemGPT: LLMs as Operating Systems, arXiv 2310.08560 treats the context as scarce main memory, backed by two external levels (*recall* of recent history and searchable *archival*), with the **agent** paging data via tool calls ("context page faults"). Decision: what to evict and what to fetch is decided by the agent, not by a fixed RAG pipeline.

- **The canonical taxonomy** — CoALA, arXiv 2309.02427 separates **episodic** memory (past experience), **semantic** memory (knowledge of the world/user) and **procedural** memory (skills/code), plus working memory. Decision: at write time, decide *which kind* of memory that fact is — each kind is retrieved differently. The memory mechanisms survey, arXiv 2404.13501 (later ACM TOIS) organizes the subsystem by *sources* · *forms* · *operations* (writing, management/consolidation, reading) — budget effort per operation, not just for the search index.
- **The recall formula** — Generative Agents, arXiv 2304.03442 (UIST '23) stores observations in a dated *memory stream* and retrieves by a composite score of **recency × importance × relevance** (exponential recency decay, importance scored by an LLM (Large Language Model), relevance by embedding). It is the concrete formula a harness should implement to rank what re-enters the context — and it introduces **consolidation by reflection** (synthesizing high-level reflections from clusters of observations).
- **Controlled forgetting** — MemoryBank, arXiv 2305.10250 (AAAI '24) decays/reinforces each memory's strength via an Ebbinghaus curve (elapsed time × access frequency), keeping the store bounded. Decision: unused memory is a pruning candidate — *usage tracking* is what closes the loop.
- **Memory as learning** — Reflexion, arXiv 2303.11366 (NeurIPS '23) converts outcome feedback into verbal self-reflection, persisted in an episodic buffer and reinjected on the next attempt — improving without updating weights. And recent architectures (A-MEM, arXiv 2502.12110; Mem0, arXiv 2504.19413) treat writing as an *extract* → *consolidate* → *link* pipeline, with the memory network self-organizing (Zettelkasten-style). Bridge to ch. 16 (self-improvement).

(Full bibliography and pointers: `livro/bibliografia.md`.)

Industry sources

- **Session as a durable event log** — Manage sessions (Claude Code): each session is continuously written to disk as **JSONL** per project (one line per message/tool-use/metadata); `--continue` resumes the most recent one in the directory, `--resume` opens a picker. Decision: "resuming" is **restoring complete state** (tool calls, results, permission mode, active goal), not text replay — the harness owns a private durable log, not a stable public schema.
- **Workspace reversal as a separate track** — Checkpointing (Claude Code) captures the code state before each prompt; `/rewind` restores code, conversation **or** both (100 recent checkpoints, cleaned up with the session). The Agent SDK's file-checkpointing exposes this as a reusable primitive. Decision: undoing the *code* is a separate store from undoing the *conversation*, linked by the prompt index.
- **Durable memory as files with precedence** — How Claude remembers your project: the CLAUDE.md hierarchy (managed policy → user → project → local), the `#` shortcut to append a memory line, `/memory` to edit. Decision: cross-session memory is **markdown in precedence tiers** (the most specific wins) — versionable, auditable, scoped; reread at launch as always-on context.
- **The memory tool (beta) and "assume interruption"** — Memory tool: the model requests operations (`view/create/str_replace/…`) on a `/memories` directory that persists across conversations, but execution is **client-side** — your app implements the storage (and the protection against path traversal, size limits, expiration). The system injects "ASSUME INTERRUPTION: your window may be reset at any moment". Paired with context management (context editing evicts stale pairs from the window; the

memory tool persists outside it) — two levels: short-term hygiene + external long-term store. Decision: for long-running agents you need both; the window is ephemeral, `/memories` is the source of truth (the pattern from the essay harnesses for long-running agents: a structured progress log, read at the start and updated at the end of each session).

- **Memory ≠ RAG** — the distinction became an industry thesis: Letta ("RAG is not agent memory") and AWS Bedrock AgentCore argue that RAG is *stateless* reading; memory is reading **+ write path + state management** (admission, resolution of conflicting facts, invalidation). Letta exposes self-editing *memory blocks* and **core/recall/archival** tiers (the MemGPT hierarchy as a product); mem0 routes each fact through a layer with its own lifetime; Zep/Graphiti models memory as a **bi-temporal knowledge graph** (outdated facts are *invalidated*, not deleted); LangMem/LangGraph separates **short-term (thread)** from **long-term (per-namespace store)**. Decision: you cannot "buy" memory by bolting on a vector store — you need a write/update/invalidation pipeline.
- **See also**: the living collection Awesome Harness Engineering — Memory & State gathers more consultable resources for this dimension (patterns, articles, and implementations), curated by problem.

The state of the art

1. Three layers, three champions — and no vector database

The problem's three layers got different champions in the cohort: **session state** (database durability — opencode with SQLite + replayable events; Codex with per-turn rollout jsonl; OpenHands with event-stream), **long-term memory** (relevance + format rigor — OpenHarness with versioned memdir, `relevance.py` and `usage.py`; Hermes with `MEMORY.md`/`USER.md` + `session_search`), **workspace state** (git-based reversal — Aider and gemini-cli). And the finding that persists: **none of the code harnesses uses a vector database** for memory. In the code domain, versionable markdown beat embeddings — because code memory needs a *write path* (the agent edits the file) and auditability, exactly what the "memory ≠ RAG" thesis predicts.

2. The recall formula and forgetting moved from paper to code

OpenHarness's `relevance.py` + `usage.py` is the practical instance of the Generative Agents stream: it selects by relevance what enters the context and marks usage — unused memory becomes a pruning candidate (MemoryBank's forgetting curve, in practice). Hermes formalizes **active maintenance**: a single tool edits `MEMORY.md`/`USER.md` with **periodic nudges** (every 10 turns), and a `session_search` (FTS5/BM25 index over the session SQLite, with discovery/recall/summarization modes) provides **cross-session recall** — MemGPT's archival layer built on textual search, not vectors.

3. Reversibility became a primitive — and it changes the risk calculus

Workspace checkpointing stopped being a feature and became a primitive: **Aider** pioneered it years ago (git-native state: atomic auto-commit per round, `aider_commit_hashes`, `/undo`, `.aider.chat.history.md`), **gemini-cli** consecrated it (`/restore`, `/rewind` of the disk via git snapshots), and Claude Code exposes it as checkpointing with separate tracks for code and conversation. The design

consequence is the most interesting part: **an agent whose actions are reversible changes the risk calculus of everything else** — permissions can be looser when undoing is cheap (ties into ch. 07).

4. Pluggable providers and the harness as a memory server

The emerging frontier: memory as a pluggable service. Hermes already accepts external providers (Honcho, mem0, supermemory) behind its layer; products like Letta/mem0/Zep position themselves as the "universal memory layer" consumable by any harness. The design tension for the next rounds: keep memory as a **local versionable file** (auditable, portable, dependency-free) or outsource it to a managed store (bi-temporal graph, scale). In code, the file still wins; outside it, the pendulum is less clear.

> **EXECUTIVE SUMMARY**
>
> What is most modern: the OS-tiers frame (RAM ↔ recall ↔ archival) with the agent paging; recall by recency×importance×relevance with usage-based forgetting; workspace reversal as a primitive that loosens permissions; and the hard memory × RAG distinction (write path + invalidation). **What to steal:** persist the session as a durable event log (resuming = restoring state, not replay); treat memory as versionable markdown with usage tracking; separate the code-reversal track from the conversation; and, for long-running agents, write a durable progress log assuming the window can vanish at any moment.

Hands-on — harness-zero, step 4

Step 4 (`harness-zero/etapas/04-sessoes/`) gives harness-zero persistence: an **SQLite adapter** behind a `StorePort` stores messages and tool calls as typed rows, and `/resume` restores the complete state of a previous session (not just the text). Faithful to hexagonal *by refactoring*: the pain that gives birth to the port is reopening the process and losing the conversation. Completeness exercise: persistence covers the *happy path*; you add a minimal `USER.md`/`MEMORY.md` read at the start and a progress log updated at the end — the "assume interruption" pattern in its simplest form.

Check your understanding

1. Why is agent memory not the same thing as RAG, and what does that explain about choosing versionable markdown over a vector database in the code harnesses? (Memory = retrieval + write path + state management/invalidation; code needs an auditable write path.)
2. You have 10,000 memories and room for 20 in the context. Which score do you use to choose, and how do you decide what to prune over time? (Recency × importance × relevance; pruning by lack of use — the forgetting curve.)
3. Your agent gained workspace checkpointing with `/rewind`. Which decision *from another dimension* does that let you loosen, and why? (Permissions — the risk calculus drops when undoing is cheap; ch. 07.)

Appendix A — How each repository handles memory and state

> Per-harness evidence, with paths — online supplement, expanded each round.

opencode (round 1) — state as a database

Persistence in **SQLite via Drizzle** (`packages/core/database`, `core/session/sql.ts`): sessions, messages and parts are typed rows. Sessions have a `parentID` (hierarchy for subagents), support revert (`session/revert.ts`) and **sharing** (`share/`, `sync/`). V2 (`CONTEXT.md`) takes the design to "data infrastructure": a durable prompt inbox, replayable events with cursors (`sessions.events({sessionID, after})`), context snapshots persisted across restarts. Round 1's most robust state model — the harness as a distributed system with durable state.

gemini-cli (round 1) — the reversible workspace

Long-term memory in the `GEMINI.md` files themselves (`save_memory` tool, global in `~/.gemini` + project index, with auto-memory tested in evals). The distinctive feature is **git-based checkpointing** (`services/gitService.ts` + `chatRecordingService.ts`): workspace snapshots before edits, enabling `/restore` and `/rewind` — undoing the agent's changes on disk, not just in the conversation — plus `/resume`.

OpenHarness (round 1) — memory as files, with discipline

`src/openharness/memory/` (13 modules): persistent memory in markdown (`MEMORY.md`/per-project memdir) with **versioned schema, atomic file-locked writes and signatures**. `relevance.py` selects what enters the context; `usage.py` marks usage (unused memory is a pruning candidate). Sessions persisted with rich metadata (`services/session_storage.py`): permission mode, read-file state, invoked skills, compaction checkpoints. Resume via `-c/--continue`, `-r/--resume`, `/resume`.

Aider (round 2) ⭐ git-native state — the reversal pioneer

`aider/repo.py`: **atomic auto-commit per round** with an LLM-generated message, configurable authorship attribution, `aider_commit_hashes` tracking what the AI did, `dirty_commit` isolating pending changes. `/undo`, `diff` and `blame` become the memory interface; complemented by `.aider.chat.history.md` and `--restore-chat-history`. **Anticipated by years** the "git checkpoint" that gemini-cli and Claude Code consecrated.

Hermes (round 2) ⭐ multi-layer memory with cross-session recall

`MEMORY.md` (agent notes) + `USER.md` (user profile) edited by a single tool with **periodic nudges** (every 10 turns); pluggable external providers (**Honcho, mem0, supermemory**); and **`session_search`** — an FTS5 index over the session SQLite with three modes (discovery/BM25, windowed recall, LLM summarization) for cross-session recall. MemGPT's archival layer on textual search.

Codex CLI (round 2) — per-turn rollout jsonl

Each turn is persisted as **rollout jsonl** (recoverable); `SessionTask` (Regular/Review/Compact/UserShell) organizes the task machine. Durable, resumable session state integrated into the loop (`core/src/session/`).

OpenHands (round 2) — persisted event-stream

`openhands/app_server/event/` persists each `Event` as JSON per conversation, with pagination, filters and trajectory export. The control plane consumes/persists events; the action-observation loop runs in the SDK. Event sourcing as the state's backbone.

OpenClaw (round 2) — session lanes and workspace files

Runs serialized per *session lane* with a file-based write-lock between processes; workspace files (`MEMORY.md`, `USER.md`, `IDENTITY.md`…) injected with budgets (20k chars/file, 60k total) and marked truncation. Per-channel conversation persistence.

ohmo (round 2) — session/memory backends as plugins

Implements OpenHarness's `SessionBackend` and `MemoryCommandBackend` as first-class plugins (without touching the core), plus a **multi-session pool** (`RuntimeBundle` per `session_key`, recreated when the cwd changes). Proof that the app/engine boundary was designed.

IronClaw (round 2) — resumable state via checkpoints

Resumable state via **checkpoints**; a Queued→Running→Blocked→Completed state machine with **leases/heartbeats** and "one active run per canonical thread". The `LoopExit` carries only durable references — the loop never mutates state; the `LoopExitApplier` validates host-owned evidence before applying.

n8n (round 2) — the workflow engine's memory

Memory via *memory sub-nodes* (`contextWindowLength` window, `maxTokensFromMemory` cutoff); workflow state persisted by the engine across executions. Short by nature — event-triggered executions do not accumulate long context (compaction score 1, by design).

Frameworks (frameworks round)

LangGraph: **checkpointer** (short-term, thread-scoped) + per-namespace **store** (long-term cross-thread); LangMem: semantic/episodic/procedural memories as tools; Agents SDK and CrewAI: session/short-term state with persistence hooks. The short × long term distinction is a framework primitive — what the code harnesses implement by hand, the frameworks expose as an API.