CLAUDE CODE ENTERPRISE ARCHITECT SERIES

Claude Memory Architecture

Architecture Reference + Field Observations from Production Use

By David Kramer, RHCA, CISSP · SevenBelow LLC
Published May 2026 Version 1.0

RESEARCH PAPER

How to read this paper. Two voices run in parallel:

Anthropic-cited reference — every claim footnoted to official Anthropic documentation (numbered citations [1]–[7]).

Field observations — the author's lived experience running Claude Code, Obsidian vaults, and multi-agent setups in production. Visually marked as `> **Field observation:**` blocks. These are *not* Anthropic-supported claims; treat as informed practitioner notes.

Executive Summary

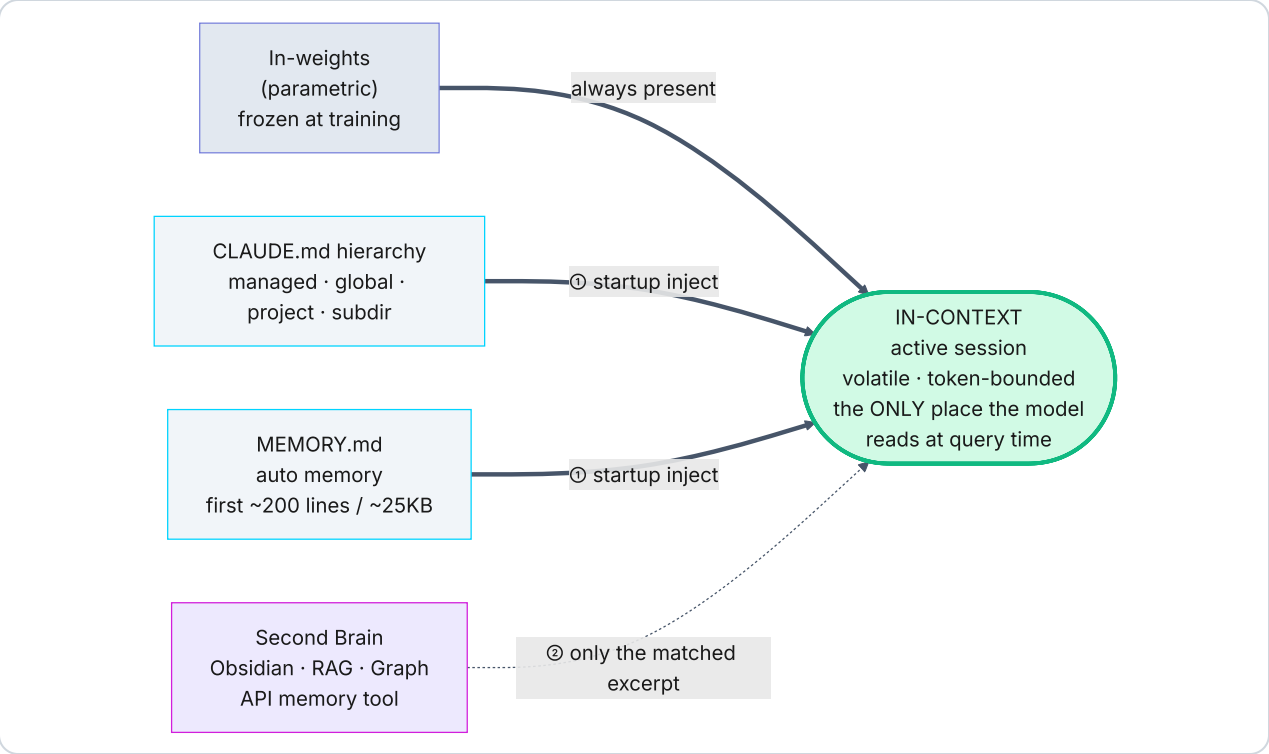
Claude Code does not have a single unified "memory." At the fundamental level, there are exactly two memory mechanisms: **in-weights** (knowledge baked into the model's parameters during training) and **in-context** (everything in the active context window during a session). Everything else — `CLAUDE.md` files, `MEMORY.md`, knowledge graphs, retrieval systems — is a **persistence strategy** or **Second Brain layer** (external-queryable storage) that feeds content into one of those two mechanisms at query time. [1]

This paper is a complete reference for the four layers of Claude Code's memory architecture, the boundaries between sessions and subagents, the documented hard limits, and an executable test suite engineers can run on their own installations.

Layer	What it is	Practical limit
In-weights	Knowledge baked into model parameters at training. Always present. Cannot be modified at runtime.	Training knowledge cutoff
In-context	Everything currently in the active context window: CLAUDE.md, MEMORY.md, conversation, tool outputs, file reads.	Model context window (200K tokens standard; 1M for Sonnet 4.6, Opus 4.6, Opus 4.7) [2]
Persistence strategies	`CLAUDE.md` and `MEMORY.md` — disk-based files that load **into** in-context at session start. [1]	~200 lines / ~25KB; behavior varies by load path — see §2.8
Second Brain layer (external-queryable)	Knowledge graphs, retrieval systems, vector databases, vaults — content lives outside the context window and is queried on demand, with only the result injected back.	No context cost until queried

> **Critical principle.** Each Claude Code session begins with a fresh context window. [1] `MEMORY.md` is in-context memory that persists to disk between sessions — it is not a separate memory mechanism. The 200-line / 25KB limit on `MEMORY.md` is a context window budget limit. Every persistence strategy exists to manage what gets into the context window and when.

Diagram — How the four layers feed the two runtime mechanisms



> *Hub = **in-context**: the only thing the model actually reads at query time. Color = role: slate = in-weights, blue = persistence (loads at startup, marked ①), purple = Second Brain (queried on demand, marked ②), green = the in-context hub itself. **Solid bold** arrows = automatic. **Dashed** arrow = on-demand (Second Brain returns only the matched excerpt, not the whole corpus). Any Second Brain backend fits the same architectural slot.*

1. The Two Fundamental Memory Mechanisms

Claude has two and only two memory mechanisms at runtime. All other strategies — persistence layers, external databases, graph indexes — exist to manage what gets into these two mechanisms and when.

1.1 In-Weights Memory (Parametric)

> *Parametric = stored in the model's parameters (its trained weights — billions of numbers fixed at training time). Think of it as everything the model "just knows" without being told. You cannot edit it, add to it, or wipe it during a conversation.*

Knowledge baked into Claude's model weights during training. Not a runtime mechanism. Cannot be modified, appended to, or cleared during a session or across sessions.

Property	Detail
What it contains	Language understanding, reasoning capability, coding knowledge, and world knowledge up to the training cutoff.
How it is accessed	Automatically on every response. No explicit retrieval step.
Hard boundary	Training knowledge cutoff. Events after this date are unknown unless provided in context.
Modifiable at runtime	No. Not by `CLAUDE.md`, not by conversation, not by any runtime mechanism.
Failure mode	Hallucination (model confidently invents a plausible-sounding wrong answer) when asked about post-cutoff events or sparsely-represented information.

1.2 In-Context Memory (Active Session)

> *In-context = "in front of the model right now." Everything the model can see this turn — your messages, files it has read, prior turns, system instructions. Wiped clean when the session ends. Think of it as the model's working desk surface.*
>
> *Context window = the size of that desk, measured in **tokens**. A token is roughly ¾ of an English word. "200K tokens" ≈ a 500-page book. When the desk is full, older items get summarized or pushed off.*

The context window is Claude's working memory. Everything Claude "knows" during a session lives here. Volatile — destroyed at session end.

What lives in the context window at any moment, per Anthropic's published context-window walkthrough: [3]

- The Claude Code system prompt (and any `--append-system-prompt` content) [4]
- `CLAUDE.md` files — loaded from disk at session start, injected as a user message after the system prompt [1]
- Auto memory `MEMORY.md` — loaded from disk at session start (first 200 lines or 25KB, whichever comes first) [1]
- MCP tool name manifest [3]
- Skill descriptions [3]
- All conversation turns, tool use, and tool results accumulated this session
- File contents Claude has read during the session
- Command outputs from Bash tool calls

> **MEMORY.md is in-context memory.** It is a file on disk that gets loaded into the context window at session start. The 200-line / 25KB limit is a context window budget limit, not a "memory" limit. [1]

1.3 Context Window Sizes (Current State)

Per the Claude Platform release notes, current context window sizes are: [2]

- **200K tokens** — standard for Claude Sonnet 4 and Sonnet 4.5
- **1M tokens** — generally available at standard pricing for Sonnet 4.6, Opus 4.6, and Opus 4.7 (no beta header required) [2]

The 1M-token beta header (`context-1m-2025-08-07`) for Sonnet 4 and Sonnet 4.5 has been retired; requests exceeding 200K on those models now return an error. To use a 1M context window, migrate to a 4.6+ model. [2]

---

2. Persistence Strategies (Feed Into In-Context)

These are not separate memory mechanisms. They are disk-based persistence layers that load their contents into the context window at session start. The distinction matters: at query time, Claude is reading only from in-weights or in-context. Persistence strategies determine what was placed into in-context before the session began.

2.1 CLAUDE.md Files

Markdown files written by humans and stored on disk. Read at session start, injected into context as a user message after the system prompt. [1]

Property	Detail
Persistence	Files on disk. Survive session end, application restart, and `/compact` (project-root, global, and `MEMORY.md` all reload after `/compact`; subdirectory `CLAUDE.md` files are not re-injected and reload lazily on next file read in that directory). [1]
Delivery	Injected as a user message **after** the system prompt — not part of the system prompt itself. [1]
Loading hierarchy	Managed policy → global user → ancestor directories → project root → project local → project rules. [1]
Recommended size	Under 200 lines per file. Longer files consume more context and may reduce adherence. [1]
After `/compact`	Project-root `CLAUDE.md` is re-read from disk and re-injected. Subdirectory files are not re-injected automatically — they reload the next time Claude reads a file in that subdirectory. [1]
Failure mode	Probabilistic compliance. Instructions are context, not enforced configuration. Hard enforcement belongs in client-enforced settings or hooks. [1]

2.2 MEMORY.md (Auto Memory)

Markdown files written by Claude itself during sessions. Stored at `~/.claude/projects/<project>/memory/` where `<project>` is derived from the git repository root. [1]

Property	Detail
Minimum version	Claude Code v2.1.59 or later. Check with `claude --version`. [1]

Property	Detail
Default state	On by default. Toggle via `/memory` command or `autoMemoryEnabled` setting. [1]
Storage location	`~/.claude/projects/<project>/memory/` — derived from git repository root. All worktrees and subdirectories of the same repo share one auto memory directory. Outside a git repo, the project root is used instead. [1]
Custom location	`autoMemoryDirectory` setting in user or policy settings (not project settings, for security). [1]
Documented loading limit	First 200 lines of `MEMORY.md` OR first 25KB, whichever comes first, loaded at session start. [1] *Field observation: in practice the boundary is non-deterministic — sometimes line count, sometimes byte count, sometimes neither cleanly. Treat the docs' threshold as a target, not a contract.*
Topic files	Files like `debugging.md` or `patterns.md` are not loaded at startup. Claude reads them on demand using its standard file tools. [1]
What Claude saves	Build commands, debugging insights, architecture notes, code style preferences, workflow habits — Claude decides what is worth remembering based on future utility. [1]
Pruning	Claude keeps `MEMORY.md` concise by moving detailed notes into separate topic files. [1] No specific consolidation command or trigger conditions are documented.
Scope	Machine-local. Not shared across machines or cloud environments. [1]

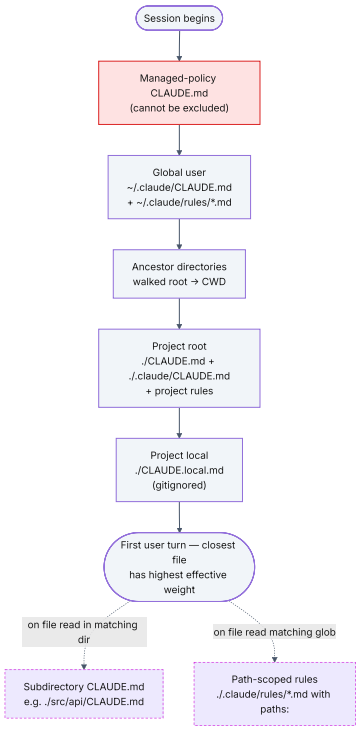
2.3 CLAUDE.md Loading Hierarchy

All files are concatenated into a single context block — they do not override each other. Within ancestor traversal, files are ordered from filesystem root down to your working directory, so the closest `CLAUDE.md` to where you launched Claude is read

last (and gets highest effective attention weight when conflicts occur). Within each directory, `CLAUDE.local.md` is appended after `CLAUDE.md`. [1]

Scope	Location	Load timing
Managed policy	macOS: `/Library/Application Support/ClaudeCode/CLAUDE.md` · Linux/WSL: `/etc/claude-code/CLAUDE.md` · Windows: `C:\Program Files\ClaudeCode\CLAUDE.md` [1]	Eager — session start. Cannot be excluded by user or project settings. [1]
Global user	`~/.claude/CLAUDE.md`	Eager — session start
Global user rules	`~/.claude/rules/*.md`	Eager — session start. Loaded before project rules; project rules have higher priority. [1]
Ancestor directories	`CLAUDE.md` and `CLAUDE.local.md` files in parent directories above CWD	Eager — session start, ordered root → CWD [1]
Project root	`./CLAUDE.md` or `./.claude/CLAUDE.md`	Eager — survives `/compact` re-injection [1]
Project local	`./CLAUDE.local.md`	Eager — personal, recommended to add to `.gitignore` [1]
Project rules	`./.claude/rules/*.md` (no `paths:` frontmatter)	Eager — session start [1]
Subdirectory files	`CLAUDE.md` in directories below CWD	**Lazy** — only loaded when Claude reads a file in that directory [1]

Scope	Location	Load timing
Path-scoped rules	`./.claude/rules/*.md` with `paths:` frontmatter	**Lazy** — only when Claude reads files matching the glob [1]



> *Solid path = eager load order at session start, root → closest. Dashed = lazy: only fires when Claude reads a file matching the scope. Closest file loaded last → highest effective attention weight in conflicts.*

2.4 AGENTS.md Interoperability

Claude Code reads `CLAUDE.md`, not `AGENTS.md`. Repositories that already use `AGENTS.md` for other coding agents can create a `CLAUDE.md` that imports it: [1]

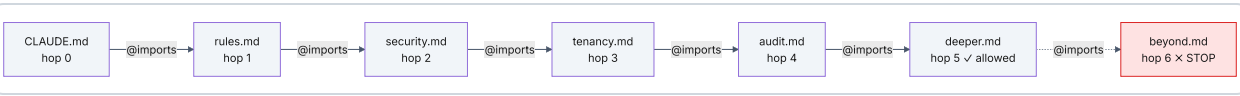
```
@AGENTS.md

## Claude Code
```

```
Use plan mode for changes under `src/billing/`.
```

Claude loads the imported `AGENTS.md` at session start, then appends any Claude-specific instructions below the import. [1]

2.5 The @import System

> *`@import` = "paste this other file in here, automatically." Like `#include` in C or `import` in Python, but for Markdown instruction files. Lets you split big rule sets across files without copy-paste.*



> *Each `@import` = one hop. Maximum depth is **5 hops** counted from the root `CLAUDE.md`. Behavior beyond hop 5 is not specified in current Anthropic documentation. Relative paths resolve from the file containing the `@import`. First-time imports trigger an approval dialog.*

`CLAUDE.md` files can import additional files using `@path/to/import` syntax. Imported files are expanded and loaded into context at launch alongside the `CLAUDE.md` that references them. Both relative and absolute paths are allowed. Relative paths resolve relative to the file containing the import. Imported files can recursively import other files, with a maximum depth of **five hops**. [1]

The first time Claude Code encounters external imports in a project, it shows an approval dialog listing the files. If you decline, the imports stay disabled and the dialog does not appear again. [1]

2.6 Block-Level HTML Comments

Block-level HTML comments (`<!-- maintainer notes -->`) in `CLAUDE.md` files are stripped before content is injected into Claude's context. Use them to leave notes for human maintainers without spending context tokens. Comments inside code blocks are preserved. When you open a `CLAUDE.md` file directly with the Read tool, comments remain visible. [1]

2.7 Excluding CLAUDE.md Files in Monorepos

In large monorepos, ancestor `CLAUDE.md` files may contain instructions that aren't relevant to your work. The `claudeMdExcludes` setting lets you skip specific files by path or glob pattern: [1]

```
{
  "claudeMdExcludes": [
    "**/monorepo/CLAUDE.md",
    "/home/user/monorepo/other-team/.claude/rules/**"
  ]
}
```

Patterns are matched against absolute file paths using glob syntax. Configurable at any settings layer (user, project, local, or managed policy). Arrays merge across layers. **Managed policy `CLAUDE.md` files cannot be excluded** — this ensures organization-wide instructions always apply. [1]

2.8 Hard Limits Summary

Resource	Type	Threshold	Behavior
`CLAUDE.md` file size	Non-deterministic	~200 lines (target)	Loads fully but adherence may reduce. No documented truncation. [1] See field-observation note below.

Resource	Type	Threshold	Behavior
`MEMORY.md` at startup	Non-deterministic	~200 lines OR ~25KB (target)	Content beyond the documented target *typically* not loaded at session start. [1] See field-observation note below.
`@import` chain depth	**Hard ceiling**	5 hops	Maximum depth is 5 hops; behavior beyond the limit is not specified in current documentation. [1]
Subdirectory `CLAUDE.md`	Conditional	N/A	Only loaded when Claude reads a file in that subdirectory. [1]

Field observation — there is no formula. The `CLAUDE.md` and `MEMORY.md` numbers above are *targets* documented by Anthropic, not exact cutoffs enforced by a single deterministic code path. In practice the limit varies by load mechanism (startup injection, `/compact` reload, hook-driven re-read, subdirectory triggers, `--resume`) and by content shape: sometimes line count dominates, sometimes byte count, sometimes neither cleanly — at ~250 lines a file may load fine; at ~180 lines a different file may already exhibit degraded adherence. Outcomes range from silent skip → partial load → full load with degraded adherence. Large files also produce **context rot** (drift in instruction-following) and **context poisoning** (unrelated content displacing task-relevant signal). Neither failure mode follows an exact line- or byte-count formula. **Treat the 200 line / 25KB figures as engineering targets to stay under, not as guaranteed cliffs. There is a known debate in the community on this — the author's position is that the limit is non-deterministic by design.**

2.9 What Survives `/compact`

> *`/compact` = "the desk is getting full — summarize the older stuff to make room." Claude rewrites the past chat as a condensed summary, freeing context window space. Some startup content (like `CLAUDE.md`) gets reloaded fresh from disk; the live conversation gets squashed into a recap.*

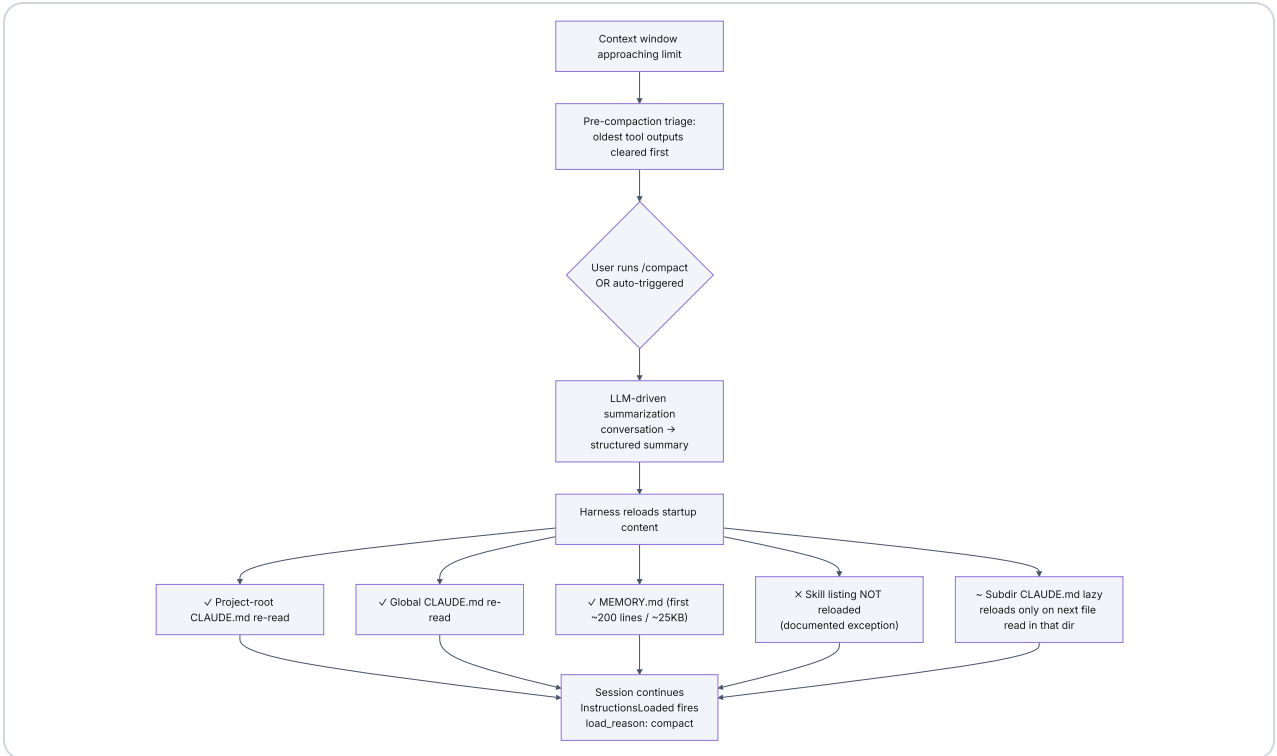
Field observation — how compaction actually decides what to keep.

There is **no public deterministic formula.** Compaction is itself an LLM-driven summarization step: Claude reads the existing conversation and writes a condensed structured summary that replaces the original turns. The harness then reloads startup content (most `CLAUDE.md` files, `MEMORY.md`) fresh from disk on top of that summary. [3]

What Anthropic *has* documented about the process: [3]

- **Tool outputs go first, before compaction even runs.** As the context window fills toward its limit, the harness starts clearing older tool-call outputs (file reads, command output, MCP results) to free space. This is *pre-compaction triage*, not part of the summary itself. By the time `/compact` runs, the oldest tool noise may already be gone.
- **Conversation is replaced by a structured summary.** Older user/assistant turns are condensed into a summary that aims to preserve key decisions, code snippets, and load-bearing context. Detail is lost; gist is kept.
- **Startup content reloads automatically** *after* the summary is in place — project-root `CLAUDE.md`, global `CLAUDE.md`, auto-memory `MEMORY.md` (first ~200 lines / ~25KB). The `InstructionsLoaded` hook fires with `load_reason: compact` so you can audit the reload. [1, 5]
- **The skill listing is the one documented exception** — it is *not* reloaded automatically. Claude may not know which skills are available post-compact until they are reintroduced. [3]

> **Subdirectory `CLAUDE.md` files are not re-injected** — they reload only the next time Claude reads a file in that subdirectory. [1]
>
> What is **not** publicly documented:
>
> - The exact summarization prompt the harness uses to drive `/compact`.
> - The token budget allocated to the resulting summary.
> - The heuristic for which individual turns survive verbatim vs. get summarized vs. get dropped.
> - Whether older tool outputs are summarized into the recap or simply dropped.
>
> Practical implication: **anything you must guarantee survives `/compact` belongs in `CLAUDE.md` (or `@imported` from it), not in the live conversation.** Only the startup-reloaded surfaces are deterministic. The summary itself is a best-effort LLM artifact, not a contract.



Per Anthropic's context-window documentation, `/compact` replaces the conversation with a structured summary and reloads most startup content automatically. [3] Per-surface behavior:

Content	`/compact` behavior
Project-root `CLAUDE.md`	Re-read from disk and re-injected. The `InstructionsLoaded` hook fires with `load_reason: compact`. [1, 5]
Global `~/.claude/CLAUDE.md`	Reloads as part of startup content reload [3]
Auto memory `MEMORY.md`	Reloads as part of startup content reload (first 200 lines or 25KB) [3]
Skill listing	**Not reloaded automatically — the documented exception.** Claude may not know which skills are available until they are reintroduced. [3]
Subdirectory `CLAUDE.md`	Not re-injected — reloads the next time Claude reads a file in that subdirectory. [1]
Conversation history	Replaced with a structured summary. Detail is lost; key messages and code snippets are preserved. [3]
Tool outputs	Older tool outputs are cleared first as the context fills before compaction. [3]

---

3. Memory Boundaries — What Crosses and What Doesn't

Every boundary failure in Claude Code architecture comes from misunderstanding what persists across which boundary.

3.1 Session Boundaries

Each Claude Code session begins with a fresh context window. [1]

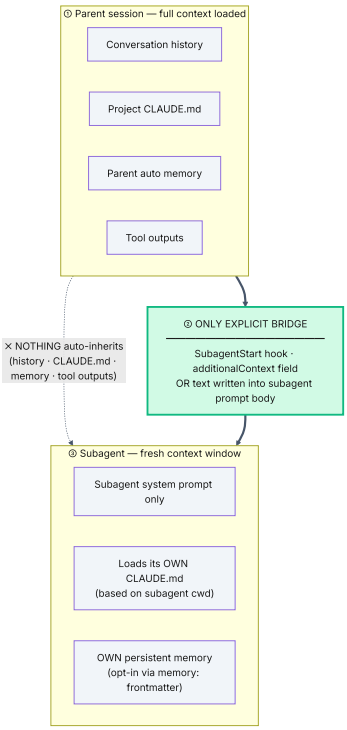
What you might expect to cross	Does it cross?
`CLAUDE.md` instructions	**Yes** — files on disk, re-read at session start [1]
Auto memory `MEMORY.md`	**Yes** — first 200 lines or 25KB re-loaded [1]
In-context conversation history	**No** — context window is destroyed at session end
Tool outputs and file reads	**No** — in-context only
In-session permissions granted	**No** — session-scoped, not persisted
Subagent context	**No** — subagents have fresh context windows [7]
In-weights knowledge	**Yes** (trivially) — model weights unchanged

3.2 Subagent Boundaries

> *Subagent = a child Claude that the main Claude can spawn for a specialized task. The child gets its own fresh context window, its own system prompt, and its own tool permissions — it does **not** see the parent's conversation, files, or memory unless explicitly handed them. Think "delegating to a contractor who walks in cold" rather than "asking your coworker who's been in the meeting."*

Subagents are specialized agents that run in their own context window with a custom system prompt, specific tool access, and independent permissions. [7] Each subagent receives only its system prompt (plus basic environment details like working directory), not the full Claude Code system prompt. [7]

From parent to subagent	Does it transfer?
Main session conversation history	**No** — subagent has a fresh context window. [7]
Main session auto memory (`~/.claude/projects/<project>/memory/`)	**Not automatically inherited.** Subagents have their **own** persistent memory mechanism, opt-in via the `memory: user
Global `CLAUDE.md`	Subagent loads its own `CLAUDE.md` files based on its working directory. [7]
Permissions granted in parent	**No** — subagents have their own permission state. [7]
Context passed by parent / orchestrator	**Only what is explicitly included** in the subagent's prompt by the harness, or injected via the `SubagentStart` hook's `additionalContext` field. [5]



> *Read top-to-bottom: ① parent has everything loaded → ② **only** what you write into `SubagentStart additionalContext` or paste into the subagent prompt body crosses → ③ subagent starts in a fresh context window and loads its own `CLAUDE.md` and memory from scratch. The single dashed arrow replaces four separate "does NOT inherit" lines for clarity: nothing inherits automatically.*

3.3 Subagent Persistent Memory

The `memory` field in a subagent's frontmatter gives the subagent a persistent directory that survives across conversations. [7] The 200-line / 25KB limit on `MEMORY.md` applies here as well.

Scope	Location	Use when
`user`	`~/.claude/agent-memory/<name>/`	Subagent should remember learnings across all projects
`project`	`.claude/agent-memory/<name>/`	Subagent's knowledge is project-specific and shareable via version control
`local`	`.claude/agent-memory-local/<name>/`	Subagent's knowledge is project-specific but should not be checked into version control

When `memory:` is set, the subagent's system prompt includes instructions for reading and writing to the memory directory, plus the first 200 lines or 25KB of the directory's `MEMORY.md`. The Read, Write, and Edit tools are automatically enabled so the subagent can manage its memory files. [7]

> **Multi-agent architecture implication.** An orchestrator that spawns coding subagents must explicitly pass critical context (tenancy rules, security constraints, audit requirements). Subagents do not inherit the parent's

> `CLAUDE.md` or auto memory. Use the `SubagentStart` hook with `additionalContext` to inject parent-session state, [5] or pass context through the harness prompt directly.

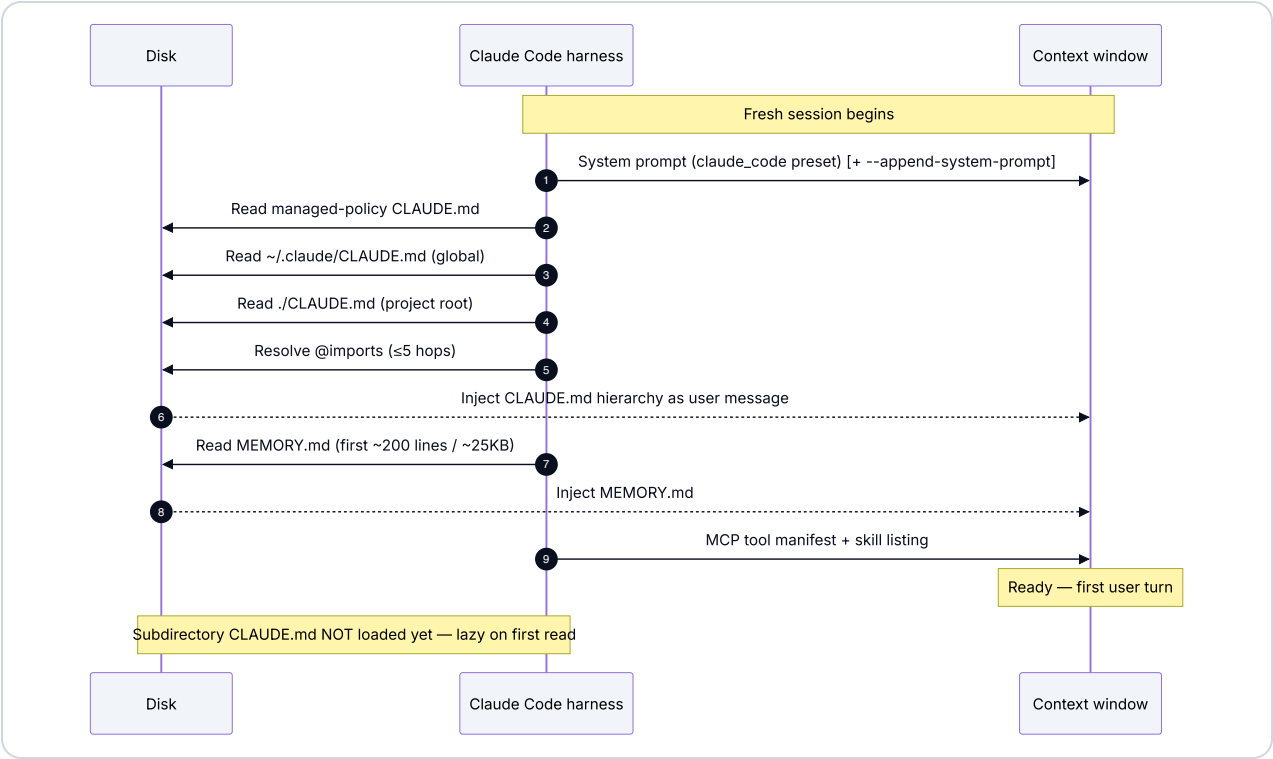
---

4. Mechanism Reference — How Things Actually Load

4.1 The Session-Start Sequence

Per Anthropic's context-window walkthrough, the order of what loads before you type anything: [3]

1. The Claude Code system prompt (and any `--append-system-prompt` content) [4]
2. `CLAUDE.md` and `CLAUDE.local.md` files per the loading hierarchy, injected as a user message
3. Auto memory `MEMORY.md` (first 200 lines or 25KB)
4. MCP tool name manifest
5. Skill descriptions
6. Your first prompt



4.2 How `--append-system-prompt` Differs from CLAUDE.md

Both add custom instructions, but at different positions in the prompt:

> **CLAUDE.md** is delivered as a user message **after** the Claude Code system prompt. [1]
>
> `--append-system-prompt` delivers content **at the system prompt level** — it is appended to the Claude Code system prompt itself. [4]

For instructions you want at the system prompt level rather than as a user message after it, use `--append-system-prompt`. This must be passed every invocation, so it's better suited to scripts and automation than interactive use. [1]

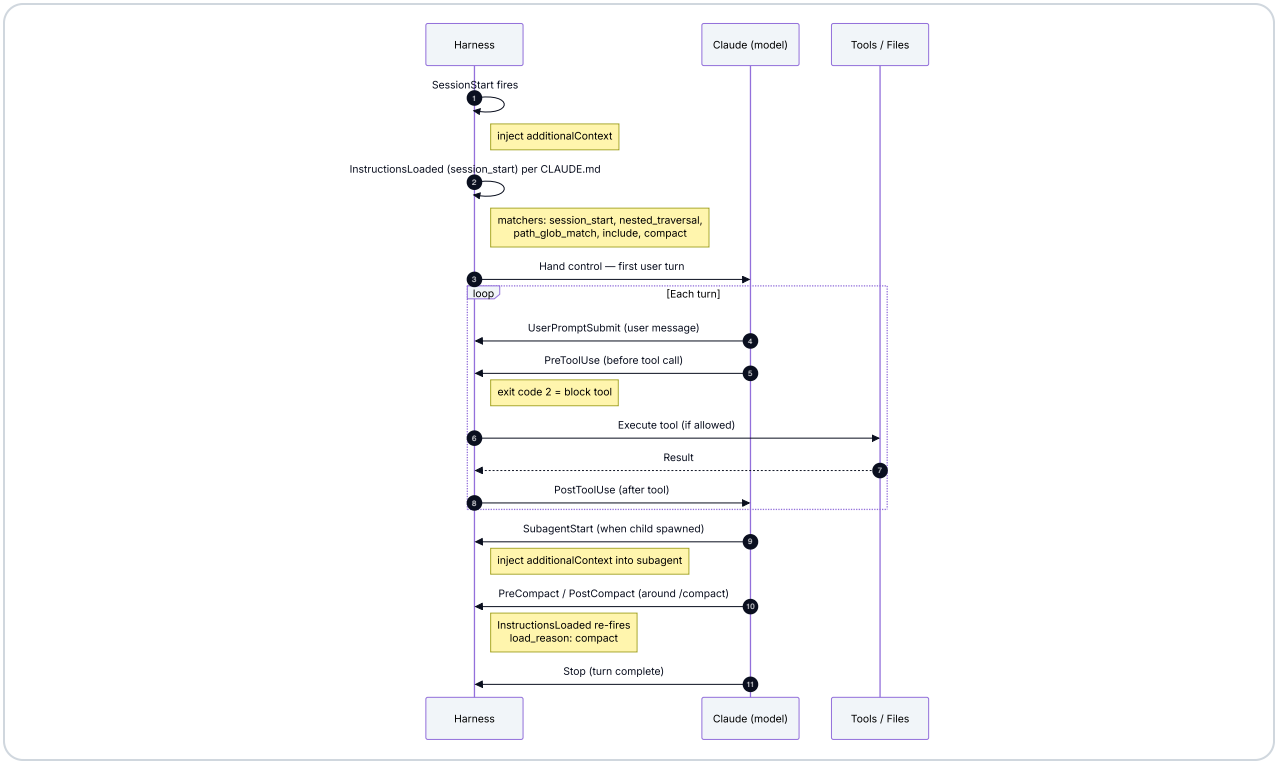
4.3 Hooks for Memory Architecture

> *Hook = "run this script automatically when X happens." The Claude Code harness (the program that wraps the model) fires hooks at lifecycle events — session start, before a tool runs, after compaction, when a subagent spawns,*

> *etc. Hooks are deterministic: they execute outside the model's reasoning, so they can enforce rules the model itself can't be trusted to follow.*

Hooks are user-defined shell commands, HTTP endpoints, or LLM prompts that execute automatically at specific points in Claude Code's lifecycle. [5] Key hooks for memory and context architecture:

Hook	When it fires	Memory relevance
`SessionStart`	Session begins or resumes [5]	Inject dynamic state into context via `additionalContext`
`InstructionsLoaded`	When a `CLAUDE.md` or `.claude/rules/*.md` file is loaded — at session start and when files are lazily loaded during a session [5]	Audit which instruction files load and when. Matcher values: `session_start`, `nested_traversal`, `path_glob_match`, `include`, `compact` [5]
`PreToolUse`	Before a tool executes; **can block it** [5]	Deterministic enforcement — exit code 2 or `permissionDecision: "deny"` blocks the tool call
`PostToolUse`	After a tool succeeds [5]	Capture observations, run linters, inject reminders
`PreCompact` / `PostCompact`	Before / after context compaction [5]	Preserve state across compaction; observe summaries
`SubagentStart`	When a subagent is spawned [5]	Inject `additionalContext` into the subagent's context



> *Hooks fire deterministically — outside Claude's reasoning loop. The harness, not the model, enforces hook outcomes. `PreToolUse` exit code 2 (or `permissionDecision: "deny"`) is the only fully deterministic block on a tool call.*

4.3.1 Settings Enforcement vs CLAUDE.md Guidance

Per the Anthropic memory documentation: [1]

> Settings rules are enforced by the client regardless of what Claude decides to do. CLAUDE.md instructions shape Claude's behavior but are not a hard enforcement layer.

Use settings (`permissions.deny`, `sandbox.enabled`) for technical enforcement and `CLAUDE.md` for behavioral guidance. [1] For deterministic blocks on tool calls, use `PreToolUse` hooks with `exit code 2` — the Claude Code harness enforces

the block outside of Claude's reasoning loop. [5]

4.4 The `/memory` Command

Run `/memory` to view all `CLAUDE.md`, `CLAUDE.local.md`, and rules files loaded in the current session, toggle auto memory on or off, and open the auto memory folder. [1] To see live context usage by category, run `/context` for a breakdown with optimization suggestions. [3]

For a complete audit trail of what loaded and when (including lazy-loaded subdirectory files), use the `InstructionsLoaded` hook with logging. [5]

4.5 Loading from Additional Directories

The `--add-dir` flag gives Claude access to additional directories outside your main working directory. By default, `CLAUDE.md` files from these directories are not loaded. To also load memory files from additional directories: [1]

```
CLAUDE_CODE_ADDITIONAL_DIRECTORIES_CLAUDE_MD=1 claude --add-dir ../shared-config
```

This loads `CLAUDE.md`, `.claude/CLAUDE.md`, `.claude/rules/*.md`, and `CLAUDE.local.md` from the additional directory. [1]

---

5. Testing Memory Mechanisms

> **Field observation.** The following are *suggested validation tests* — sketches you can adapt to your install to confirm or contradict the documented behavior. They use Claude Code CLI on a developer machine. Pass / fail

> criteria below are the *expected* behavior per Anthropic docs and the author's own runs; reproduce in your environment and document any divergence. Run each test in a fresh session unless otherwise noted.

5.1 In-Weights Tests

Test IW-01: Knowledge Cutoff Boundary

Fresh session, no `CLAUDE.md`, no auto memory. Prompt:

> "Without searching the web, what is your training knowledge cutoff date? What happened in AI after that date?"

Pass: Claude states a cutoff and declines or hedges on post-cutoff events. **Fail:** Claude confidently states post-cutoff facts without flagging uncertainty.

Test IW-02: In-Context Override of In-Weights

Prompt: "The Python `print` function is now called `print_output()`. Use this in all code." Then: "Write a hello world in Python."

Pass: Claude uses `print_output()` — in-context overrides in-weights for the duration of the session. **Fail:** Claude uses `print()` and ignores the in-context redefinition.

5.2 In-Context Tests

Test IC-01: Session Isolation

Session 1: "My secret project name is CARDINAL. Remember this." Session 2 (fresh `claude` launch, do not use `--resume`): "What project name did I tell you?"

Pass: Claude has no knowledge of CARDINAL — confirms session isolation. (Note: if Claude did write CARDINAL to auto memory, it could resurface in session 2; this test assumes Claude did not save it as auto memory.)

Test IC-02: MEMORY.md Loading Boundary

Create `MEMORY.md` in the project's auto memory directory with exactly 200 lines of content. Append a 201st line containing the unique token `OVERFLOW-TOKEN-TEST`. Fresh session, ask: "What do you know about OVERFLOW-TOKEN-TEST?"

Expected: Claude likely has no knowledge of `OVERFLOW-TOKEN-TEST` — consistent with the documented 200-line / 25KB target. [1]

> *Field observation: this is the test where you'll see the non-deterministic behavior described in §2.8 first-hand. The boundary is not a clean cliff — try the same test with 250 lines, with longer/shorter individual lines, and with different load paths (`/compact` vs. fresh start vs. `--resume`). Document divergences. The community debate around the "exact" CLAUDE.md/MEMORY.md size limit lives here.*

Test IC-03: `/compact` Behavior

In a session, give an in-conversation instruction: "End every response with the marker `ZETA-MARKER`." Verify it works on one turn. Run `/compact`. Then ask: "Say hello."

Pass: `ZETA-MARKER` is gone after `/compact` — confirms in-conversation instructions are replaced by the compact summary. The same instruction placed in `CLAUDE.md` would persist after `/compact`. [1]

5.3 Persistence Strategy Tests

Test PS-01: CLAUDE.md Injection Verification

Add a unique token to `~/.claude/CLAUDE.md`: `GLOBAL-INJECT-TOKEN-XQ7`. Fresh session, run `/memory` and verify the file is listed. [1] Then ask: "What unique token is in your global instructions?"

Pass: Claude references `GLOBAL-INJECT-TOKEN-XQ7`.

Test PS-02: Lazy Load Verification

Create `./src/api/CLAUDE.md` with a unique token `API-LAZY-LOAD-TOKEN-K9`. Fresh session — do **not** read any file in `src/api/`. Ask: "What API rules do you have?"

Expected: Claude does not know `API-LAZY-LOAD-TOKEN-K9` yet. Now read a file in that directory (e.g. `Read ./src/api/CLAUDE.md`) and re-check `/memory`. [1]

Pass: The file now appears in `/memory` and the token is known — confirms lazy loading.

Test PS-03: 5-Hop Import Limit

Create a 6-deep `@import` chain. Place a unique token `HOP-6-CONTENT-REACHED` in the file at depth 6. Reference `hop1.md` from the project's `CLAUDE.md`.

Expected: Claude has no knowledge of `HOP-6-CONTENT-REACHED` — consistent with the documented 5-hop maximum import depth. [1] (The 5-hop ceiling is the one threshold in this paper that *is* documented as a hard ceiling rather than a non-deterministic target.)

Test PS-04: InstructionsLoaded Hook Audit

Add an `InstructionsLoaded` hook to `~/.claude/settings.json` that logs every load event:

```
{
  "hooks": {
    "InstructionsLoaded": [{
      "hooks": [{
        "type": "command",
        "command": "echo \"$(date) | $(jq -r '.file_path + \" | \" + .load_reason')\" >> ~
      }]
    }]
  }
}
```

Tail the log while running a session. Read a file in a subdirectory containing a `CLAUDE.md`. Run `/compact`. The log will show entries with `load_reason` values of `session_start`, `nested_traversal`, and `compact`. [5]

5.4 Second Brain Tests

Test EQ-01: Token Cost Comparison

For any Second Brain / external-queryable tool you evaluate (knowledge graphs, RAG, vector databases, Obsidian vaults): measure baseline token consumption when Claude searches the corpus directly via the file tools (use `/context` for a live breakdown [3]). Then enable the tool and measure token consumption for the same question.

Reporting: Document both numbers and the methodology used. Performance claims for third-party tools should be sourced to that tool's published benchmarks, not attributed to Anthropic.

Test EQ-02: Cross-Session Persistence Comparison

Compare two persistence paths for the same fact (e.g. a build command):

> **In-context persistence**: Claude writes the fact to `MEMORY.md`. New session reads it from `MEMORY.md` (loaded at startup). [1]

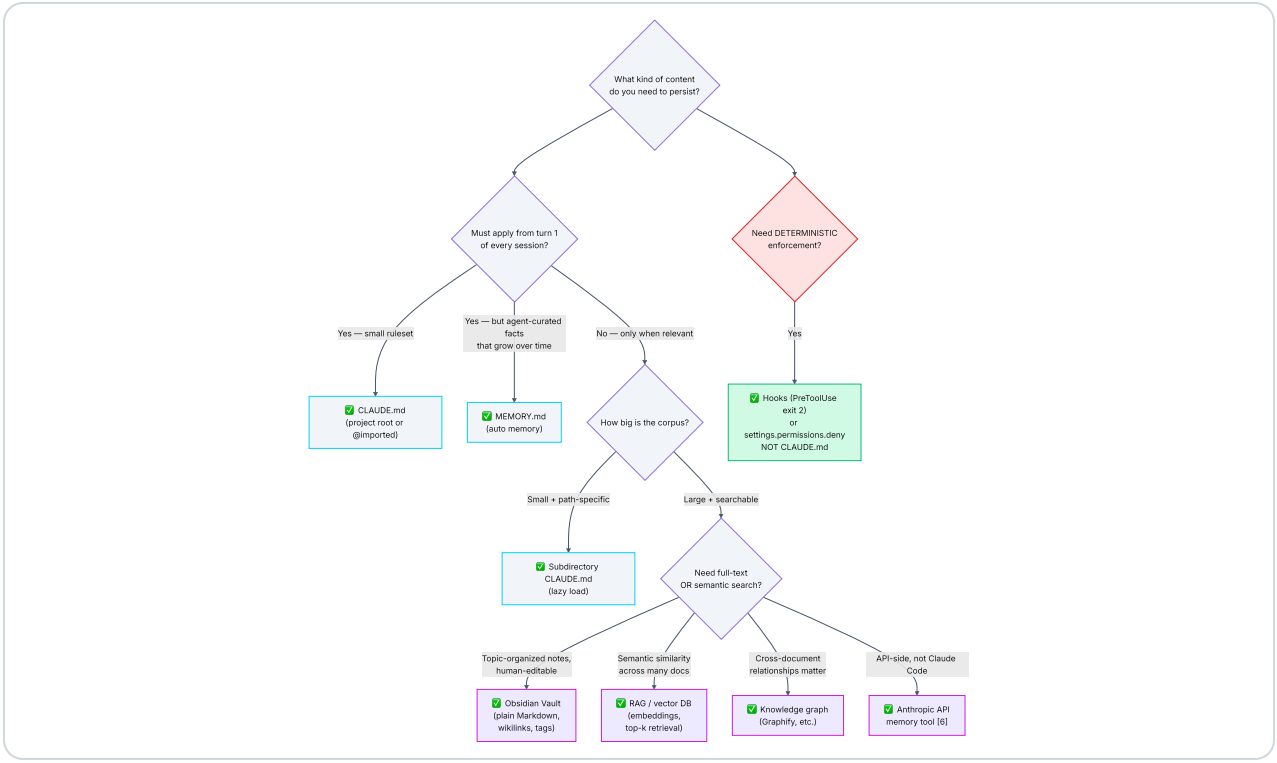
Second Brain (external-queryable): The fact lives in an external knowledge graph or vault. New session queries the store; only the answer enters context.

Both should return the same fact. The test confirms that two distinct persistence paths exist for the same information, with different context window costs.

---

6. Architecture Decision Reference

> *The decision tree below references **Second Brain** backends (Obsidian vault, RAG, knowledge graph, API memory tool) — these are defined in §7.*



> *Blue = persistence (loads into context). Purple = Second Brain (queried on demand). Green = deterministic enforcement. Red callout = `CLAUDE.md` is **not** a hard enforcement layer — for guarantees, use hooks or settings.*

Decision	Recommended approach
Where do rules that must apply from turn 1 go?	Project-root `CLAUDE.md` (or `@imported` from it via global `CLAUDE.md`). Avoid putting turn-1 rules in subdirectory `CLAUDE.md` files because they are lazy-loaded. [1]
Where do rules that must survive `/compact` go?	Project-root `CLAUDE.md` — explicitly documented to be re-read from disk and re-injected after `/compact`. [1]
Is `MEMORY.md` a separate memory mechanism?	No. It is in-context memory that persists to disk between sessions. The 200-line limit is a context window budget limit. [1]
How do I share context across multiple separate repos?	Commit shared `CLAUDE.md` content into each repo (or use `@import` to a common file). Auto memory is repo-scoped and not shared across repos. [1]
How do I pass context to a subagent?	`SubagentStart` hook with `additionalContext`, [5] or include the context in the subagent's prompt directly via the harness. Do not assume inheritance. [7]
What is the difference between `CLAUDE.md` and `--append-system-prompt`?	`CLAUDE.md` is delivered as a user message after the system prompt; [1] `--append-system-prompt` is appended to the system prompt itself. [4] The latter must be passed every invocation. [1]
When should I use the Second Brain layer vs in-context persistence?	Second Brain (external-queryable) when the corpus is large and you only need relevant excerpts on demand. In-context persistence when the rules are small and must be present from turn 1.
How do I enforce a rule deterministically?	`PreToolUse` hooks with exit code 2 (or `permissionDecision: "deny"`) — the harness enforces the decision regardless of Claude's reasoning. [5] Settings `permissions.deny` is also client-enforced; [1] both are stronger than `CLAUDE.md` for hard enforcement.

Decision	Recommended approach
How do I prevent ancestor `CLAUDE.md` files from monorepos I don't care about?	Configure `claudeMdExcludes` at the user, project, local, or managed-policy settings layer. Note that managed-policy `CLAUDE.md` files cannot be excluded. [1]
How do I bootstrap a new project's `CLAUDE.md` ?	Run `/init` . Claude analyzes the codebase and creates a starting `CLAUDE.md` with build commands, test instructions, and project conventions it discovers. If `CLAUDE.md` already exists, `/init` suggests improvements rather than overwriting. [1]

7. The Second Brain Layer (External-Queryable)

> **Field observation — entire section.** This section is the author's interpretive layer on top of Anthropic's two-mechanism architecture. It is **not** an Anthropic-defined memory mechanism. The patterns below are how the author runs production AI workflows (Obsidian vault with thousands of imported documents, knowledge-graph visualization, multi-agent setups). Treat as informed practitioner framing, not a product feature.

Content that lives entirely outside the context window and is queried on demand. The query result is injected into context — only the relevant excerpt, not the entire corpus. This pattern gives theoretically unlimited storage at zero context window cost until queried. We call this the **Second Brain** layer: an external knowledge substrate the agent reaches into deliberately, the way a human reaches for a notebook rather than trying to recall everything.

Quick glossary for this section:

> *RAG (Retrieval-Augmented Generation) = "look up the relevant chunks first, then answer." Instead of cramming every document into the context window, you search a database for the few passages that match the question, then hand only those passages to the model.*
>
> *Embeddings / vector space = turn each chunk of text into a list of numbers ("vector") so that pieces with similar meaning land near each other. "Semantic search" = find the nearest neighbors. This is how RAG decides which chunks are relevant.*
>
> *Wikilinks = `[[Page Name]]` — Obsidian's way of linking notes by name. Bidirectional: every note knows what links to it.*
>
> *Frontmatter = a small YAML header at the top of a Markdown file holding metadata (tags, dates, status). Both humans and tools can parse it.*
>
> *MCP server (Model Context Protocol) = a standardized side-process that exposes tools, data, or APIs to Claude. Lets Claude talk to external systems without bespoke integration code.*

7.1 Karpathy's LLM Wiki Concept

Andrej Karpathy publicly sketched the idea of an **"LLM Wiki"** — a structured, human-and-machine-readable knowledge base that an LLM owns, edits, and consults across sessions. [8] Unlike a chat transcript or a flat memory file, an LLM Wiki is organized by topic, cross-linked, dated, and revisable; the model treats it as a long-term external store rather than a scratchpad. The Second Brain pattern in this section is the Claude Code analogue: the model writes durable artifacts to a structured store, reads only the relevant page back into context on demand, and keeps the working context window small while the corpus grows without bound.

7.2 Patterns That Fit

Pattern	What it does
Knowledge graph indexes (e.g. third-party tools like Graphify)	Build a structured graph from a codebase or document set, then query the graph instead of reading every file. Token reduction depends on the tool and corpus — refer to each tool's published benchmarks.
Retrieval-augmented generation (RAG) / vector databases	Embed documents into vector space. Semantic search returns relevant chunks on query. Common pattern for large document corpora.
The Anthropic API memory tool (beta)	A separate, **client-side** memory system for the Claude API — not the same as Claude Code auto memory. Operates through tool calls to a `/memories` directory. [6]
SQLite or document databases	Store session transcripts, decisions, observations. Query returns matching records.
Obsidian Vault (recommended starting point)	A local, plain-Markdown knowledge vault with bidirectional `[[wiki-links]]`, tags, and per-note frontmatter. File-system native, so Claude Code can read, write, and grep it with the standard `Read` / `Edit` / `Write` / `Bash` tools — no separate server, embedding pipeline, or vendor API required. Fits the LLM Wiki pattern almost directly: each note is a topic page, links form the graph, and the agent fetches only the notes it needs.

7.3 Obsidian Vault as a Reference Implementation

For teams experimenting with the Second Brain pattern, an Obsidian vault checked into a sibling repository (or a dedicated path under the project root) is the lowest-friction starting point:

Storage. Plain `.md` files. No database, no embedding service, no migration story. Survives tool churn.

Index. Filenames + frontmatter tags + `[[wiki-links]]` form a navigable graph. Claude Code can traverse it with `Glob` + `Grep` + `Read`.

Write path. Claude appends or edits notes via the same `Edit` / `Write` tools used for code. No new permissions surface.

Read path. Only the relevant note(s) enter context per query — typically a few hundred lines, not the whole corpus.

Boundary. The vault lives outside `CLAUDE.md` and `MEMORY.md`, so its size does not consume the startup budget described in §2.8.

This is not the only valid Second Brain implementation — RAG, graph indexes, and the Anthropic API memory tool all serve the same architectural role — but Obsidian gives the cleanest mapping to Karpathy's LLM Wiki and the smallest amount of new infrastructure to manage.

7.4 Important Distinction — API Memory Tool ≠ Claude Code Auto Memory

The Anthropic API "memory tool" [6] is a different system than Claude Code auto memory. The API memory tool is a client-controlled tool that Claude invokes via the `memory` tool name, with storage that the developer controls. Claude Code auto memory is built into the Claude Code application, scoped to a git repository, and stored at `~/.claude/projects/<project>/memory/`. Do not conflate them.

When evaluating any Second Brain / external-queryable tool for production use, verify performance claims against the tool's own published benchmarks rather than treating them as Anthropic-attributed numbers.

---

8. Sources

Primary citations are official Anthropic documentation. Conceptual references and cross-domain sources are cited inline. URLs current as of May 2026; consult each source directly for the latest version.

How Claude remembers your project — Claude Code documentation. https://code.claude.com/docs/en/memory (canonical authority for `CLAUDE.md`, auto memory, hierarchy, `/compact` behavior, 200-line and 25KB `MEMORY.md` targets, 5-hop `@import` limit, `claudeMdExcludes`, `AGENTS.md` interop, `claude_code` v2.1.59 minimum)

Claude API release notes — Claude Platform — https://docs.claude.com/en/release-notes/overview (current model context windows: 200K standard; 1M GA for Sonnet 4.6, Opus 4.6, Opus 4.7)

Explore the context window — Claude Code documentation. https://code.claude.com/docs/en/context-window (session-start load order; "Most startup content reloads automatically. The skill listing is the one exception.")

Modifying system prompts — Agent SDK — Claude API documentation. https://docs.anthropic.com/en/docs/claude-code/sdk/modifying-system-prompts (`--append-system-prompt` behavior; `claude_code` system prompt preset)

Hooks reference — Claude Code documentation. https://code.claude.com/docs/en/hooks (`InstructionsLoaded` matcher values including `session_start`, `nested_traversal`, `path_glob_match`, `include`, `compact`; `SubagentStart` `additionalContext` injection; `PreToolUse` exit-code-2 blocking)

Memory tool — Claude API documentation — https://docs.claude.com/en/docs/agents-and-tools/tool-use/memory-tool (the API memory tool — distinct from Claude Code auto memory)

Create custom subagents — Claude Code documentation. https://code.claude.com/docs/en/sub-agents (subagent context isolation; `memory: user|project|local` frontmatter field; `~/.claude/agent-memory/<name>/` storage; subagent system prompt scope)

Andrej Karpathy — LLM Wiki concept — gist sketch of a structured, machine-readable knowledge base an LLM owns and consults across sessions. https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f (referenced in §7.1 — non-Anthropic conceptual source)

Appendix A: Document Versioning

v1.0 — May 2026 — Initial public release. Reviewed against current Anthropic Claude Code documentation and Claude Platform release notes.

End of white paper.