Agent Memory spec [draft]

Draft by Charles Packer (CEO, Letta)

Problem

- Most harnesses implement some sort of memory solution, but there is no standardization beyond “memory as markdown”
    - Some harnesses re-use `AGENTS.md` (DeepAgents), some use `MEMORY.md` (Claude Code, Codex), some use `USER.md` / `HUMAN.md` (OpenClaw, Hermes)
- Lack of **memory portability** (value for agent users)
    - Non-trivial to move “memory” from harness-to-harness (should be able to “move” a stateful agent by dragging and dropping
    - Cannot design systems/models/algorithms for memory without a standard target
- Lack of **memory consistency** (value for agent developers)
    - If you’re a developer it’s unclear how you should structure your file-based memory system (How many files? What names? Nesting / no-nesting?)

Existing memory standards

Existing standards cover some forms of memory:

- **AGENTS.md**
    - *Think of AGENTS.md as a README for agents: a dedicated, predictable place to provide the context and instructions to help AI coding agents work on your project.*
    - Support for nested AGENTS.md within large repositories
    - Project/repository-specific
- **Agent Skills**
    - Procedural memory for embedding specific expertise/capabilities into agents - must be able to define *when* to use it.
        - Skills require an explicit trigger condition (the description).
    - Can be project, environment, or agent-scoped

Current standards lack specifications for:

- Core memory (system-prompt/in-context memory)
- External memory that is *not* procedural (e.g. episodic, semantic, or other generic context owned by the agent) - including standards for how to progressively disclose external memory.

Why should an “agent memory” spec define? Key job: **progressive memory disclosure**

- Assumption: memory storage is “markdown files in a folder” (everyone does this)
    - We also expect agent memory to grow quickly over time
- What memory is loaded into the context window is incredibly important
    - E.g. a “persona” *needs* to be injected into the system prompt or it won’t condition the LLM’s outputs properly. Daily logs of activity should not be injected into the system prompt.
- Main job of the spec: make it clear what memory goes **inside** the context window (or system prompt), and what will stay **outside** the context window

What’s out-of-scope:

- Git-tracking (eg MemFS), length enforcement, storage mechanism, non-markdown files
- Specifying the structure of memory beyond what is necessary for progressive disclosure of agent-owned context

Agent Memory Specification

Memory directory

Agent Memory is a directory containing Markdown files and optional subdirectories:

```
None
memory/
├── MEMORY.md
├── persona.md
├── human.md
├── projects/
│   ├── MEMORY.md
│   └── project.md
└── notes/
    ├── MEMORY.md
    └── 2026-08-12.md
```

Agent memory consists of core (in-context) memory and external (out of context) memory:

- Top-level files are the **core memory** which is always in-context
- Memory that is not in context is collapsed into subdirectories.
    - Creating subdirectories is a mechanism for structuring progressive disclosure: placing context in a deeper subdirectory places it lower in the context hierarchy.
- Any directory (including the top level) that is part of the agent's memory must contain a `MEMORY.md`
    - A MEMORY.md file can store directory level memory context, index the current directory, or some combination of both.
    - The agent must be able to discover context throughout the entire filetree through progressive discovery of MEMORY.md files.

Valid memory configurations

Top-level `MEMORY.md` only, with collapsed logs (similar to Claude Code):

```
None
memory/
├── MEMORY.md
├── example-repository-name/
|   ├── MEMORY.md
|   └── 03082026.md
|   └── 03092026.md
```

Core memory only, with no collapsed logs (similar to Hermes/OpenClaw):

```
None
memory/
├── MEMORY.md
├── SOUL.md
├── USER.md
```

```
None
memory/
├── MEMORY.md
```

Nested subdirectories for progressive disclosure of external memory (similar to Letta Code):

```
None
memory/
├─── MEMORY.md
├─── projects/
|    ├─── MEMORY.md
|    ├─── project_1/
|         ├── MEMORY.md
```

Invalid memory configurations

No top-level MEMORY.md file:

```
None
memory/
├─── project-1/
|    ├─── MEMORY.md
|    └─── user_preferences.md
|    └─── bad_bugs.md
├─── project-2/
|    ├─── MEMORY.md
|    └─── memories_03082026.md
|    └─── memories_03092026.md
```

Harness contract

A conforming harness follows four rules:

1. **Load root Markdown**. Every `.md` file directly inside the memory root is included in the agent's context.
2. **Defer nested Markdown**. Markdown below the root is not automatically loaded into context.
3. **Surface deferred memory**. If nested memory exists, the harness gives the agent enough information to know it exists and find relevant files. This may be a root `MEMORY.md`, an injected path tree, memory search, or an equivalent mechanism. Ordinary filesystem readability without memory-specific discovery is insufficient.

4. **Support selective reads**. The agent can load individual deferred files when needed, through file tools or an equivalent memory-read interface.

Size guidance [recommendation]

Root Markdown should stay small because it is expected to be injected in the prefix on every request. Harnesses can enforce size limitations at edit time, e.g. by blocking a memory append if over a limit.

Agents should be guided to make use of nested for progressive memory disclosure and to keep the root context cost small. A good recommended target may be <30k tokens of memory for a 200k context window (<15%).

Scope

Agent Memory standardizes the directory and loading behavior. It does **not** define:

- how memory is edited or organized (beyond core/root + external)
    - specific harnesses may require specific files
- who owns or may write each file
- where memory is stored (locally vs mirror) or how it is synchronized
- whether memory is version-controlled (e.g. w/ git)
- when a running conversation refreshes the prefix after an edit

Harnesses may implement those features however they choose while preserving the directory contract above.