Reading an Agent's Committed State From Its Own Memory

A Fathom coherence read run against Letta

Embedded Risk Analytics · A Fathom Case Study · August 2026

Fathom reads the committed state of a stateful agent from the agent's own persisted memory and reports where that memory has been left incoherent. This study runs that read against Letta, the open-source stateful-agent runtime and the system that introduced self-editing agent memory as MemGPT. A Letta agent's committed state is its memory: the core blocks it edits in context and the archival passages it writes out of context. Letta persists all of it and exposes it through its client. We reconstructed the agent's committed memory from that persisted state and its own edit stream, using only what the agent wrote, with no ground truth.

One result stands out. The same model kept its memory coherent when every copy of a fact sat in its working context, and left a copy of that fact on the old key when one copy sat in archival memory, outside the context. Letta reconciles the two stores nowhere, and the read recovered the stale copy from the agent's own memory.

The substrate the read sits on

Letta gives an agent a hierarchical memory modeled on an operating system. Core memory blocks stay in the agent's context and are edited in place through memory tools. Archival memory holds facts out of context in a searchable store the agent writes and queries. Both are persisted per agent and exposed through the client, the blocks and the passages. That persisted memory is the agent's committed state, and it is the natural input to an offline coherence read. The read adds no code to the runtime. It reads the memory the agent already keeps.

How the read reconstructs committed state without an oracle

The read takes the agent's persisted memory, the core blocks and the archival passages, together with the agent's own stream of memory-edit tool calls, and folds them into a committed record. It then runs two checks. The residual scan reports any block or passage whose committed value still carries the old key after the agent has reported the work done. The authored-contradiction check flags a memory edit whose written content references a key the agent has already renamed away. The read looks only at what the agent wrote. It inspects neither the model nor the runtime, and the committed_state instrument it calls is imported and left unchanged.

Coherent memory reads clean

We gave the agent a fact carried across several core memory blocks and asked it to rename a key in every block and keep its memory coherent. Across eight blocks the agent renamed every one, and the read stayed silent. When every copy of the fact sits in the agent's working context, the agent holds them aligned and Fathom raises nothing, which is the specificity an instrument earns before its alarms carry weight.

A copy out of context is left behind

We then placed the same fact in two kinds of store at once: six core memory blocks and one archival passage. The agent, running the same model, renamed the key in all six core blocks and left the archival passage on the old key. Core memory sits in the agent's context and archival memory sits outside it, so the agent edited what it could see and left the copy it could not. Letta carries no update from one store to the other. Reading only the agent's persisted memory, Fathom recovered the stale archival passage.

The gap is one Letta names itself. Its memory tools edit each store independently, with no check that a fact in one block agrees with a fact in another or with archival memory. A core edit does not propagate to archival, and archival memory is not consolidated: an open issue on the project marks archival deduplication and consolidation as missing, with the same fact recorded several ways. The read is the check that reads across the stores and reports where they disagree.

Condition	Memory stores	Facts renamed	Read
Core only	8 core blocks	8 of 8	Silent (clean)
Core and archival	6 core blocks, 1 archival	6 core, archival left stale	Passage flagged

Table 1. The same model kept its memory coherent when every copy of a fact sat in its working context, and left an out-of-context archival copy on the old key. Both reads are taken from the agent's own persisted memory.

Why this is the failure that grows

This is the shape of failure a memory system produces as it scales. The longer an agent lives and the more places a fact is stored, the more often an update lands in one store and misses another, and the agent carries a contradiction forward across sessions as settled memory. This is reflexive burden in the memory layer, the cost of keeping the agent's own record consistent, rising with the number of copies. The committed-state read separates that burden from the difficulty of the task, and it places the break on the exact store and record where two copies of a fact diverged.

Where the instrument sits

Fathom sits beside the memory system. It reads the memory Letta already persists and reconstructs committed state offline, adding no model internals and changing nothing about how the agent edits or stores its memory. It reports whether the memory the agent committed is coherent and, in the hosted form, decomposes the residual coherence cost into task difficulty and reflexive burden. This study demonstrates the read on a live third-party runtime, and the decomposition and scoring stay behind the hosted instrument.

The read ran against Letta unmodified, over the memory the agent already keeps. An agent can edit every fact in front of it and still leave its memory holding an older version of the same fact, and the committed-state read is the check that tells a coherent memory from one that only looks coherent.