RESEARCH October 20, 2026

MemCurrent: Designing Persistent Memory for Production Agents

by The Databricks AI Research Team

AI tools have made it easy to vibe code agents. Building and maintaining an enterprise agent in production, however, remains significantly more challenging. In particular, developers are often forced to make a trade-off between quality (getting good answers, especially on long-tail tasks) and efficiency (keeping cost and latency in check at scale). Oftentimes, techniques bring gains in one dimension at the cost of another.

Memory is a promising way to improve both at the same time. It can help agents carry useful context across tasks, such as user preferences, prior corrections, metric definitions, and successful workflows. In principle, this can improve accuracy, reduce repeated errors, and avoid unnecessary rediscovery. But more history is not necessarily better memory. Supplying every past interaction is costly and can introduce stale, conflicting, or irrelevant context. The central challenge is deciding what to retain, how to maintain it, and how to measure usefulness. ← ключевые вопросы относительно памяти

In this post, we introduce **MemCurrent**, a persistent memory architecture for production agents, which now powers the new Agent Bricks Memory and Genie. Agents read and write memory as they work, and an offline "dreaming" pipeline consolidates memory to keep it consistent and fresh. On our internal benchmark, built from realistic enterprise tasks, MemCurrent triples the agent's pass rate over having no memory, while using 31% fewer tokens and running 26% faster. The rest of this post covers how we designed and evaluated it, and what we learned along the way.

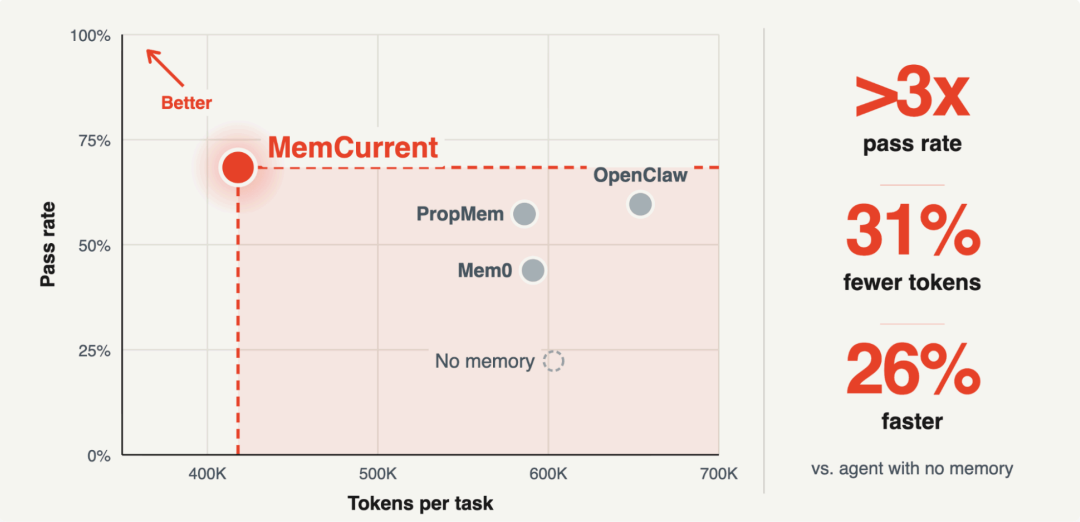


Figure 1. Pass rate vs. tokens per task on our Internal Memory Benchmark. MemCurrent is on the Pareto frontier, outperforming Mem0, PropMem, and OpenClaw memory on both dimensions.

How We Designed Memory

There are three ways to wire memory into an agent. As a **plugin** (like our earlier work on MemAlign and Memory Scaling), memory is read and written automatically at fixed points in a session. As **tools**, the agent decides what and when to read and write. As an **offline pipeline**, memory is updated in the background on a schedule or trigger, and used in later sessions. Each comes with trade-offs:

How to wire memory into agent
1) plugin
2) tool
3) offline pipeline

Memory as a ...	Triggered by	Time of computation	Pros	Cons
Plugin	Automatic	Online	• Updates take effect immediately • Predictable behavior	• Higher runtime cost and latency • Inflexible (e.g., wasting tokens even when memory isn't needed)
Tool	Agent	Online	• Updates take effect immediately • Agent has flexibility	• Higher runtime cost and latency • Quality constrained by agent capability
Pipeline	Schedule or trigger	Offline	• Lower runtime cost and latency (async batched)	• Updates arrive with a delay

Table 1. Three access patterns of memory: plugin, tool, or offline pipeline.

MemCurrent uses all three, each where it fits best.

Финская картинка, это объясняет концепцию памяти

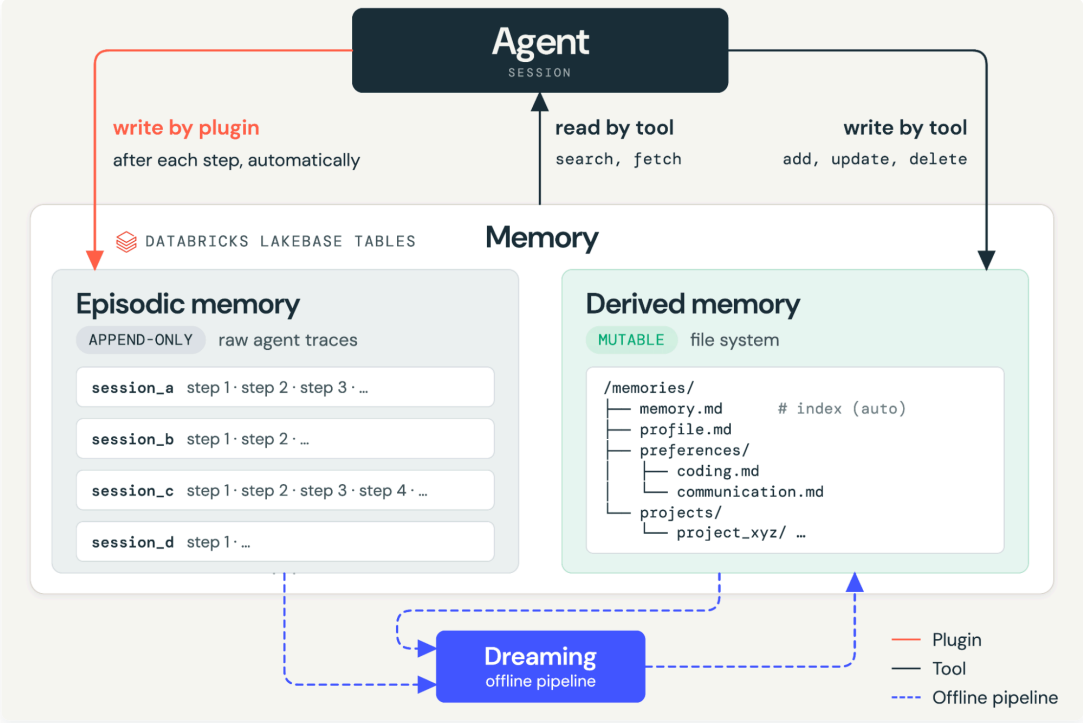


Figure 2. Overview of MemCurrent.

MemCurrent has two types of memory: **episodic memory**, which stores raw session traces; and **derived memory**, which has the distilled learnings from these traces, stored as files in a structured knowledge base. An auto-generated `memory.md` index sits at the top, describing the file structure and what each file contains.

Both episodic and derived memory live in a production-grade database that can scale to many users while delivering low-latency responses. Episodic memory is append-only, where a plugin logs each step as the session runs. Derived memory is a mutable table simulating a hierarchical file system, where writing happens in two ways. Agent tools (`add_memory`, `update_memory`, `delete_memory`) handle explicit requests like “Never write scripts in the /tmp dir”. An offline dreaming pipeline

captures the rest, distilling implicit learnings from each new session into the relevant files. It also cleans up the whole store, splitting oversized files, merging duplicates, and resolving conflicts.

Reading works in layers. At session start, a plugin injects the memory index, so the agent knows what it knows before it begins. During the session, the agent has two tools that work across both memory types: `search_memory` returns the top matches for a query (a step in a past trace, or a line in a memory file), and `fetch_memory` returns the full entry by identifier (session ID or file path).

How We Evaluated Memory

Benchmarks

We start with three popular open-source benchmarks: LoCoMo, LongMemEval, and ConvoMem. All three test whether an agent can recall facts stated earlier in open-ended synthetic conversations.

In addition, we built an Internal Memory Benchmark from real Genie conversations, where users ask all kinds of questions about their data. As in Figure 3, we started from 12K+ sessions, mined 3K candidates containing memory-worthy signals, and drafted a follow-up question with scoring rubrics. The other sessions from the same user serve as fillers for the test case. An adversarial judge then checked each test case against 7 criteria, such as whether the knowledge is actually present, specific to the user, required to answer, natural, and verifiable. After a final manual review, the benchmark has a total of 114 test cases, with 29 users and 2,233 history sessions.

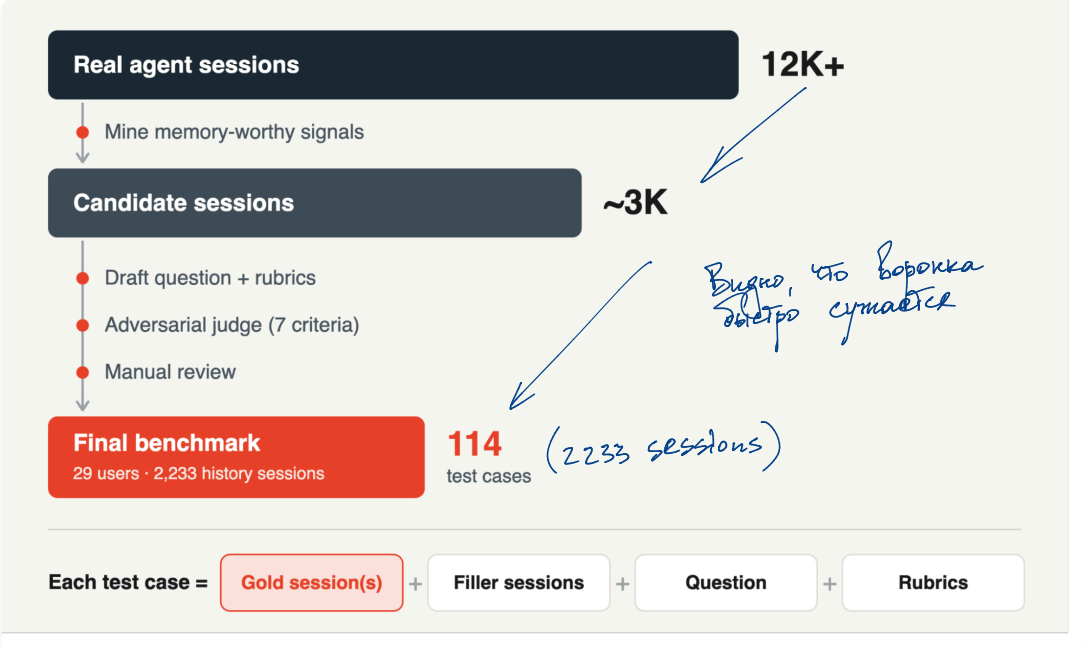


Figure 3. How we built our Internal Memory Benchmark.

Here's an example:

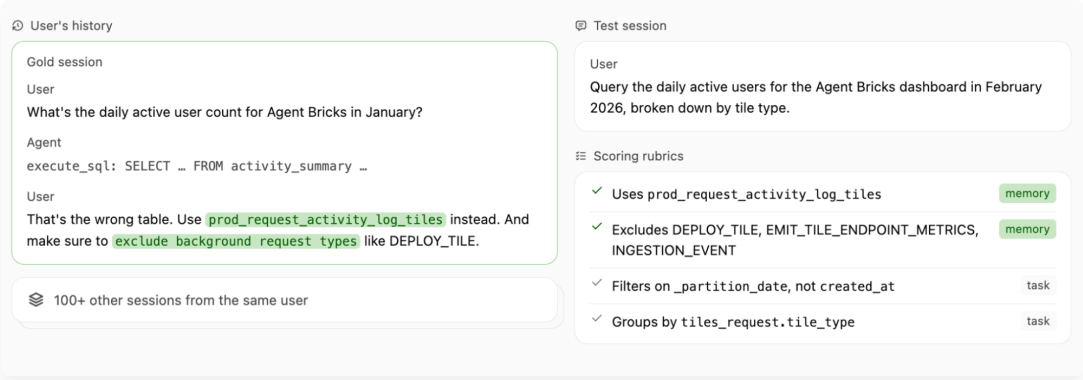


Figure 4. An example test case from our Internal Memory Benchmark. In a past session, the user corrects the agent's choice of table and tells it to exclude background requests. Later, the agent faces a different but related question, and has to find that session among 100+ others and apply what it learned. The output is scored by an LLM judge based on a list of rubrics (some memory-dependent, some task-intrinsic).

Our benchmark differs from existing ones in three ways:

1. **Enterprise domain**. Existing benchmarks are open-ended chats about the lives of simulated personas. Ours is from real work in a data workspace, involving rich enterprise knowledge like table semantics, metric definitions, and workflows.

2. **Memory feeds actions, not answers**. Finding the right string in past conversations is not enough; agents must apply retrieved knowledge to complete multi-step tasks.

3. **Diversity.** Test cases span 5 memory types (profile, preferences, domain knowledge, projects, workflows) crossed with 11 scenarios (direct instruction, changing or outdated facts, multi-session aggregation, memory-independent questions, and more).

Results

We compare MemCurrent with three open-source memory backends, Mem0, PropMem, OpenClaw memory, using the same underlying LLM.

Open-Source Benchmarks. We pair each memory backend with a simple question-answering agent, built with two agent harnesses (LangGraph and OpenAI Agents SDK). We also run two baselines: a floor with **no memory**, and a ceiling with **full context**, where every past session goes into the agent's context. Across all three benchmarks, **MemCurrent scores highest among all memory backends** and **stays on the cost-accuracy Pareto frontier**. On average, MemCurrent (88.1%) performs on par with full context (87.8%) **at ~12x fewer input tokens per question** (8.4K vs 103K). On the two benchmarks with the longest histories, it even beats full context, where dumping everything into the prompt ends up confusing the agent.

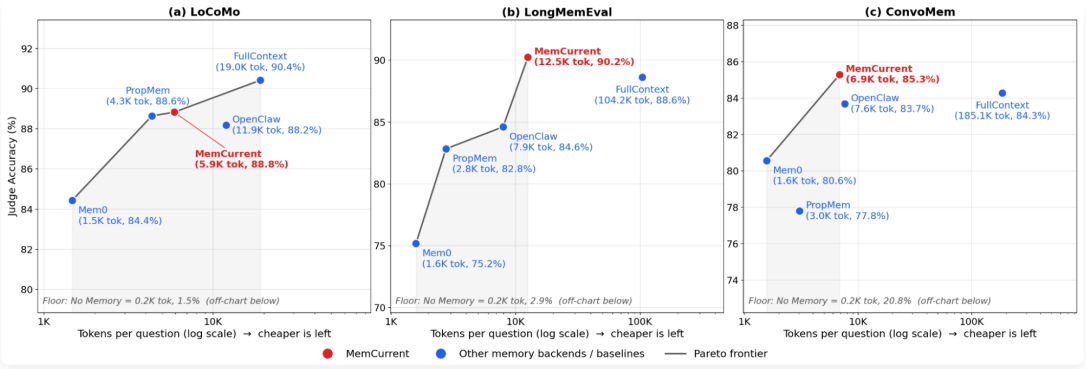


Figure 5. Cost–accuracy[1] trade-off of memory backends on (a) LoCoMo (19–32 sessions/sample), (b) LongMemEval (s split) (38–62 sessions/sample), and (c) ConvoMem (stratified subset of 500 samples, 100 sessions/sample). We use GPT 5.6 luna as the agent and memory model. All memory backends are run as a plugin, which automatically searches for related memories to the question and injects them into the agent context at session start.[2] Accuracy is the mean over the LangGraph and OpenAI Agents SDK harnesses and 3 repeats. Cost is mean input tokens per question (log scale).

Internal Memory Benchmark. On our benchmark, the gap widens. Here, each memory backend runs inside our internal Genie, the same agent the benchmark sessions came from. The histories are too long to fit in context, so our ceiling is gold injection: injecting only the gold memories into the agent's context. Among all memory backends, MemCurrent has the highest pass rate (68.4%) and the lowest latency (252s) and token usage (418K), closing over 80% of the gap between no memory (22.3%) and gold injection (78.6%). Notably, while Mem0 was also on frontier on earlier benchmarks, it trails at

43.9% here with higher latency and cost. Compared to no memory, **MemCurrent triples the agent's pass rate** (22.3% → 68.4%) **while making it 1.35x faster and 1.44x cheaper**.

На внутренней базе результаты отличные

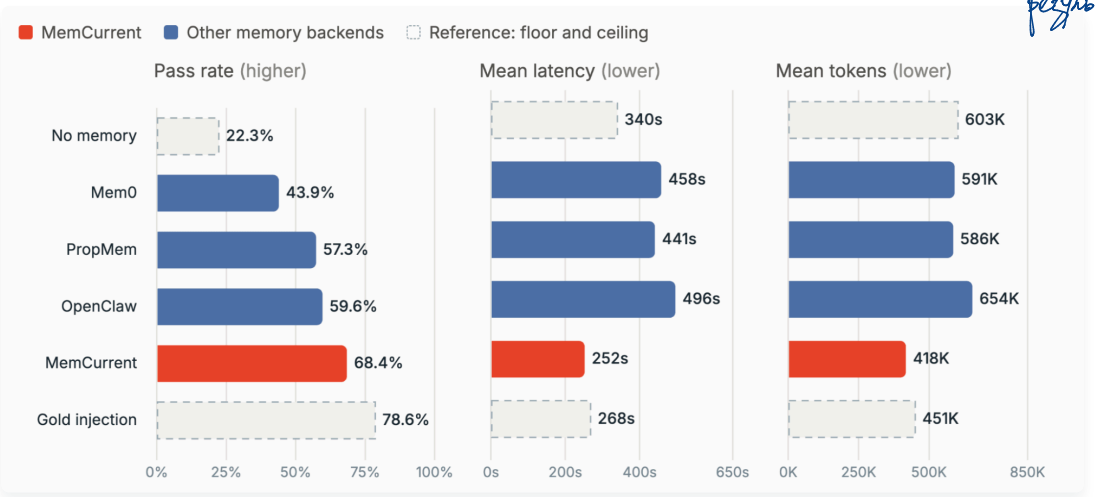


Figure 6. Pass rate, mean latency, and mean token usage per task on our Internal Memory Benchmark across 3 repeats. We use Claude Opus 4.6 as the agent model and GPT 5.6 sol as the memory model. All memory backends are provided as agent tools and a session-start plugin.

MemCurrent in Action

А дальше идёт разбор трейсов

In addition to benchmark numbers, let's examine two examples of how MemCurrent helps in practice, from internal Genie sessions.

Example 1: Memory stops an occasional hallucination mistake. In Figure 7, a user asked the agent for the top 20 users of Product X over the past two months. The agent wrote a correct SQL query, but never read the actual execution results saved to a file. It still confidently produced a table with hallucinated content. Upon user feedback, the agent saved the lesson to memory. Next time, once the query ran, the agent read the results first and returned the right table. One memory correction kept a one-off slip from becoming a habit.

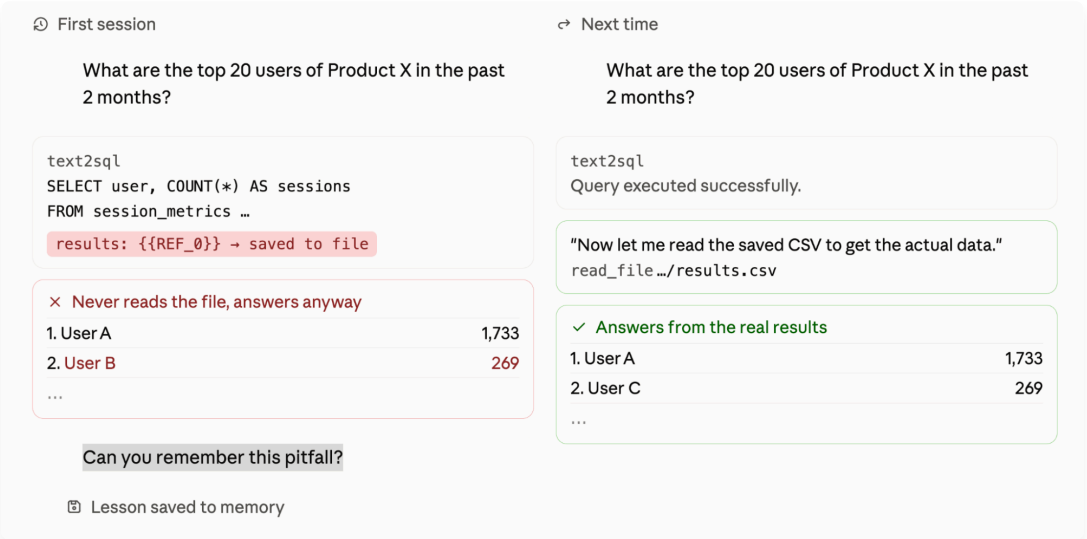


Figure 7. An anonymized example of how memory helps the agent avoid a repeated hallucination mistake.

Example 2: Memory saves computation on related tasks. In Figure 8, a user asked for "the top 5 SDP customers". The agent had to figure everything out from scratch: what "SDP" stands for, which tables held usage data, and how to join and filter them. It took 11 tool calls, 8 subagents, and nearly six minutes. The agent saved these steps as a workflow file in memory, which it could reuse on related tasks. Later, the user asked a different question: how many SDP customers had more than 5K pipeline

runs last month. Using the workflow, the agent answered with 2 tool calls and 1 subagent in 41 seconds. The question was new, but the path to the answer wasn't, and memory let the agent skip unnecessary exploration.

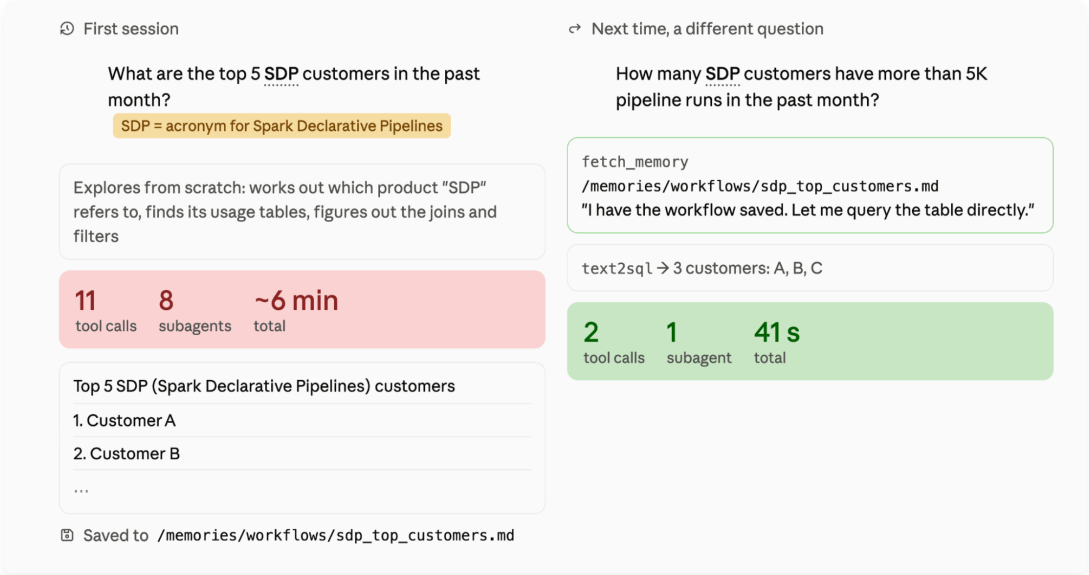


Figure 8. An anonymized example of how memory turns a 6 minute task into 41 seconds.

What Makes MemCurrent Work

To understand what accounts for MemCurrent's superior performance, we ran ablations on our Internal Memory Benchmark for two design choices: memory types and extraction strategy.

Do we need both memory types? Episodic memory alone helps moderately: 35.8%, up from 22.3% with no memory. The raw history is useful, but the agent only searches it in about a third of test cases, though it passes 60% of those when it does. Derived memory avoids this issue, as the injected memory index at session start gives the agent information about what's available in memory and when to look, lifting pass rate to 66.1%. Adding episodic memory on top gives the best result, at 68.4%. The gain is small, but it suggests raw traces can recover details that distillation drops.

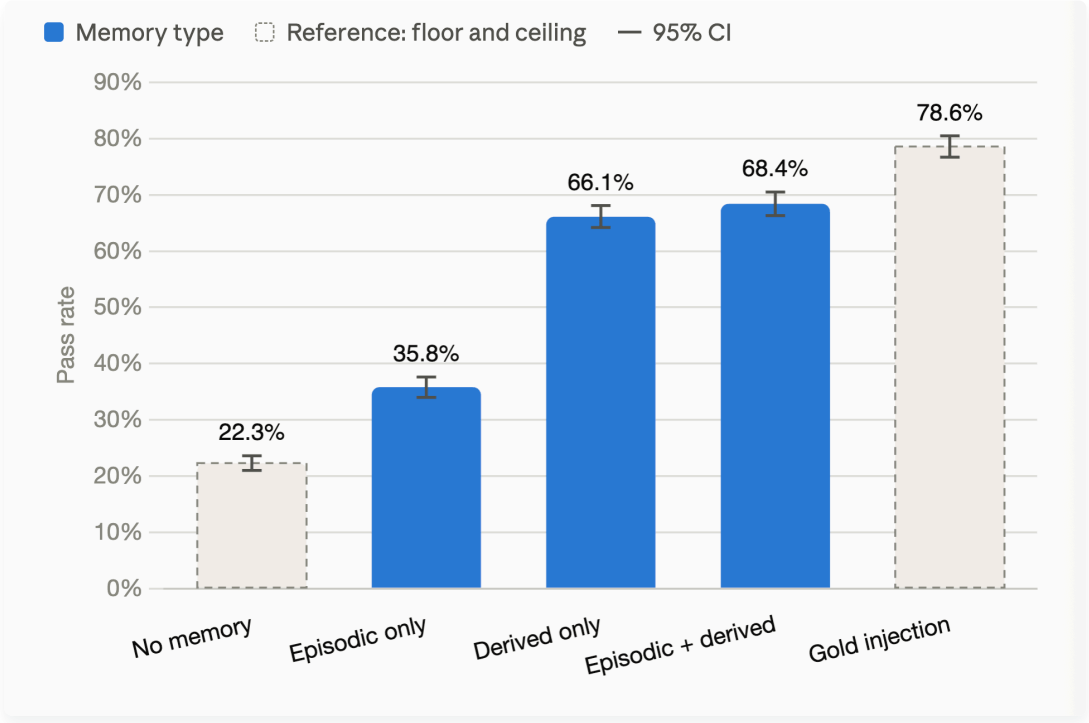


Figure 9. Pass rate by memory type.

What's a good memory extraction strategy? As shown in Figure 2, two channels extract derived memory: online tools and offline dreaming. When isolated, offline dreaming (63.5%) beats online tools (53.9%) by nearly 10 points. Their combination produces the highest pass rate at 66.1%. That supports a split of duties: dreaming handles background extraction and consolidation, while online tools make explicit requests take effect right away.

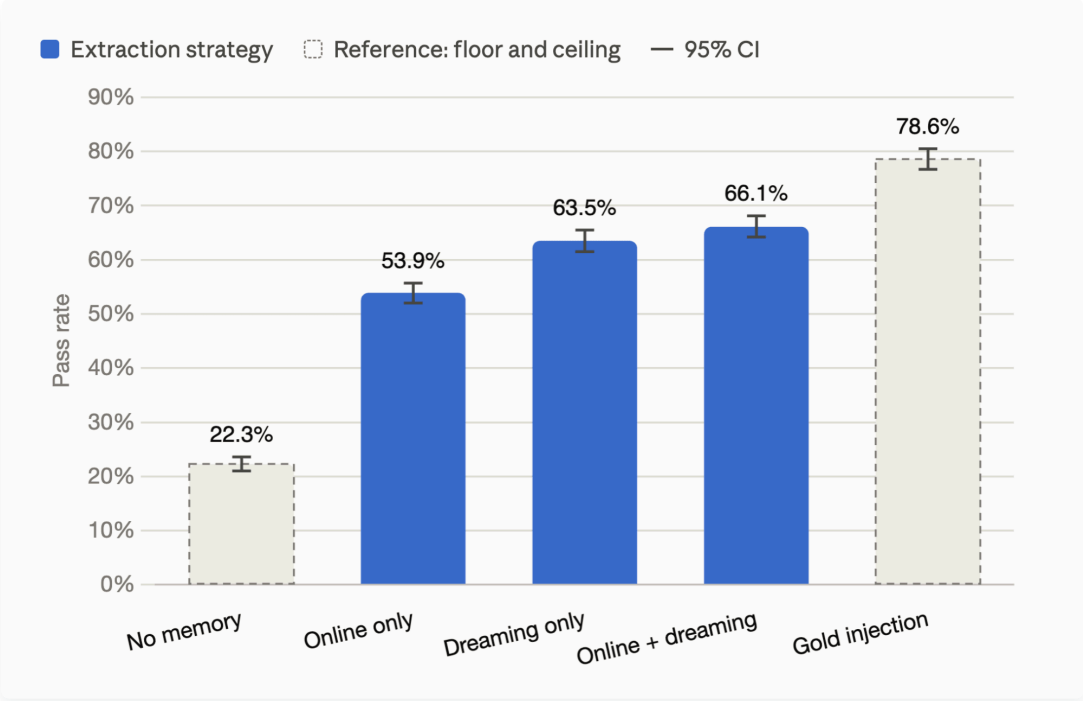


Figure 10. Pass rate by extraction strategy.

Conclusion

Memory is how an agent stops starting from zero. In this post, we introduced MemCurrent, a memory architecture for production agents that combines episodic memory and derived memory, with three access patterns: plugins to log and load context automatically, tools the agent calls on demand, and an offline dreaming pipeline that consolidates memory in the background.

Across all benchmarks we ran, MemCurrent improves quality and efficiency at the same time, the trade-off we set out to break. On open-source benchmarks, it matches full context performance at about 12x fewer input tokens. On our Internal Memory Benchmark, built from real data work, it triples the agent's pass rate over no memory while also making it faster and cheaper.

The ablations point to where the value comes from. Derived memory does most of the work, and episodic memory adds a smaller boost by recovering details that distillation drops. Combining online tools with offline dreaming makes the best extraction strategy: explicit requests take effect immediately, while dreaming distills implicit learnings and consolidates the store in the background.

Beyond the numbers, memory provides a knowledge base that isn't tied to a single agent or LLM. It can move with users to new agents or LLMs, and could seed knowledge shared across their organization, a direction we are exploring next.

Getting Started

We're bringing these findings to drive improvements across Databricks products:

For agent developers. Agent Bricks is Databricks' developer platform for building custom agents. MemCurrent is built into Agent Bricks' Memory offering, powered by Lakebase. The offering

includes:

- Session store for episodic memory
- Memory store for derived memory
- An automatic dreaming pipeline
- Tracing and evaluation of agent quality, including the impact of memory
- A CLI which helps you quickly get started with an end-to-end sample agent with all of the above in <10 minutes

Get started building your custom agent today.

For Databricks Genie users. Genie is an interface where Databricks users can converse with their data in natural language. MemCurrent is already in action behind the scenes, helping to improve answer quality while reducing response latency and cost.

Get started with Genie.

Authors: Veronica Lyu, Wenhao Zhan, Sam Havens, Ziyi Yang, Jenny Sun, Qinxin Wang, Nihit Desai, Sirui Sun, Ankit Mathur, Jane Zhang, Yi Liu, Jialu Liu, Michael Bendersky, Matei Zaharia, Xing Chen

We thank Jonathan Li, Rishabh Singh, Andy Zhang, Kenneth Choi, Will Tipton, Cassie Cao, Siddharth Murching, Sue Ann Hong, Pranav Sukumar, Neo Ni, Shanshan Zheng, Chung Wu, and our internal user study participants for their feedback and support throughout this project.

[1] The latency is similar (4-7s/question) across all memory backends, as most questions are answerable within a single hop.

[2] We find plugin-based memory sufficient for the existing benchmarks. Results from adding tools are strictly dominated and thus omitted.

RELATED TAGS: Genie and Agent Bricks

Get the latest posts in your inbox

Subscribe to our blog and get the latest posts delivered to your inbox.

Work Email *

Country *

Germany

Subscribe

By clicking "Subscribe" I understand that I will receive Databricks communications, and I agree to Databricks processing my personal data in accordance with its Privacy Policy.

View all blogs