MEMTRACE: TRACING AND ATTRIBUTING ERRORS IN LARGE LANGUAGE MODEL MEMORY SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Memory is essential for enabling large language models to support long-horizon reasoning, yet existing memory systems remain unreliable and difficult to debug. Tracing memory's dynamic evolution is crucial to understand how information is synthesized, propagated, or corrupted over time. In this work, we study the new problem of error tracing and attribution in LLM memory systems. We propose a novel framework that transforms memory pipelines into executable memory evolution graphs, enabling fine-grained tracing of operational information flow. We then construct `MemTraceBench`, a benchmark collected from representative memory systems such as Long-Context, RAG, Mem0, and EverMemOS, to systematically study memory failure modes. We further introduce an automatic attribution method that iteratively traces operation subgraphs to pinpoint the root cause of any failed case. Our analysis reveals that memory failures are systematic, stemming from operation-level issues like information loss and retrieval misalignment. Crucially, we leverage these fine-grained attribution signals to guide downstream prompt optimization, establishing a closed-loop system that automatically corrects faults and boosts end-task performance by up to 7.62 percentage points.

1 INTRODUCTION

Memory systems are a core component of large language model (LLM) agents, enabling them to evolve from isolated task solvers into stateful systems capable of long-horizon tasks and continual learning (Xu et al., 2025; Fang et al., 2025; Yang et al., 2026; Cao et al., 2025; Wang & Chen, 2025). By retaining information across interactions, updating state over time, and leveraging past experience for future decisions, memory has become widely adopted in applications such as personalized assistants and coding agents (Park et al., 2023; Li et al., 2024a; Yang et al., 2025; Xiong et al., 2025; Wang et al., 2025c;b; Wen et al., 2025). However, as memory systems become increasingly complex, a fundamental question remains underexplored: **when a memory-augmented agent fails, how can we identify where the error originates?**

Compared to prior work on diagnosing stateless agentic systems (Baker et al., 2025; Zhang et al., 2025d; Wang et al., 2026; Li et al., 2026), failure attribution in LLM memory systems presents a distinct challenge. In stateless agents, failures are often localized within the current execution trajectory, such as an incorrect tool call, retrieval result, or reasoning step. In contrast, memory-augmented agents maintain persistent states across interactions, so failures may originate from earlier memory construction, update, or deletion operations and only surface much later during retrieval or response generation. For example, a user preference may be correctly stored at first but later overwritten by an incorrect update, causing a downstream failure far removed from its origin. Such failures are difficult to diagnose from chronological traces alone. A linear execution log records operations in order, but as a flat sequence from different parts of the memory pipeline, it lacks the structure (Jiang et al., 2023) needed to show how memory variables are created, modified, overwritten, propagated, and finally used in a failed prediction. Existing memory benchmarks (Maharana et al., 2024; Wu et al., 2025; Chen et al., 2025; Bian et al., 2026) are similarly outcome-oriented: they can reveal whether a system successfully stores, retrieves, or uses relevant information, but they are not designed to recover the causal path by which a failure is introduced and propagated. This exposes a key traceability gap in LLM memory systems: failures are observable, yet the faulty operations, their introduction time, and their propagation paths remain difficult to identify.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

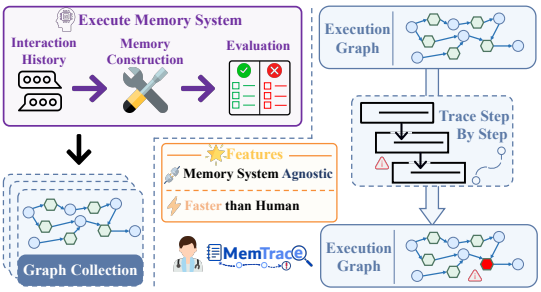


Figure 1: **Framework for automatic diagnosis of LLM memory systems.** We first execute a memory system to construct an execution graph. Given a failed case, `MemTrace` performs step-by-step tracing over this graph to locate the faulty operation. This framework is general across different memory systems and enables faster failure attribution than human experts.

To address this problem, we propose a novel framework for error tracing and attribution in LLM memory systems, as shown in Figure 1. **Our key idea is to expose memory-system execution as a unified operation-variable graph through a system-agnostic tracing toolkit.** This execution graph records memory operations and their associated variables, and connects variables through shared operations to reveal information flow during memory construction, update, retrieval, and reasoning. Unlike chronological logs, the graph explicitly captures which operations consume, modify, overwrite, or propagate each memory variable, allowing attribution to follow information dependencies across turns and sessions. Based on this representation, we introduce three contributions. First, we define a structured error taxonomy grounded in execution graph patterns. Second, we construct `MemTraceBench`, a diagnostic benchmark with human-annotated 160 real failure cases from four memory systems and three public datasets, each including question-answer (QA) pairs, execution logs, ground-truth error labels, faulty operations, and human explanations. Third, we propose `MemTrace`, an automatic attribution method that operates directly on execution graphs: given a failure case, it retrieves relevant source messages and then traces information-flow subgraphs to locate the decisive faulty operation. Extensive experiments on `MemTraceBench` show that diagnosing failures in memory systems remains challenging. However, `MemTrace` can successfully recover meaningful faulty operations and error types, and generate explanations for system debugging. Beyond error analysis, its attribution signals can guide automatic prompt optimization, improving end-task performance by up to 7.62 percentage points.

2 Tracing and Attributing Errors in Memory Systems

Automatic failure attribution consists of two steps: collecting the system's execution trace and analyzing it to localize the failure source. In this work, we use $\mathcal{M}$ to denote a non-parametric memory system that processes a trajectory τ and answers question q with a prediction $\hat{a}$ and a golden answer a. Its execution consists of memory updates $\mathcal{U}_{\mathcal{M}}$, memory reads $\mathcal{R}_{\mathcal{M}}$, and answer generation $\mathcal{Q}$. See Appendix C for the full formalization.

Background. Concretely, we instrument the source code of the memory system and execute it on a trajectory τ and question q. Whenever the system performs a memory update $\mathcal{U}_{\mathcal{M}}$, a memory read $\mathcal{R}_{\mathcal{M}}$, or an answer generation step $\mathcal{Q}$, we use a toolkit (Details in Appendix E) to automatically record the involved variables (e.g., the input question q and the predicted answer $\hat{a}$), the operations applied to them, and the dependency relations among them. This process produces an execution graph $\mathcal{G} = (\mathcal{V}, \mathcal{O}, \mathcal{E})$, where $\mathcal{G}$ is a directed acyclic bipartite graph. The node set consists of variables $\mathcal{V}$ and operations $\mathcal{O}$. Variables represent concrete artifacts produced during execution, such as raw messages, retrieved memory units, intermediate summaries, and prompts. Operations represent computation steps, such as LLM inference, tool invocation, retrieval, filtering, or parsing functions. The directed edges $\mathcal{E}$ capture information flow between variables and operations. Each operation $o \in \mathcal{O}$ takes a subset of variables as inputs, denoted as $\mathrm{In}(o) \subset \mathcal{V}$, and produces a subset of variables as outputs, denoted as $\mathrm{Out}(o) \subset \mathcal{V}$. Finally, we define a binary outcome indicator $Z(\mathcal{G}) \in \{0, 1\}$, where $Z(\mathcal{G}) = 1$ indicates that system fails to answer the question, and $Z(\mathcal{G}) = 0$ indicates success. In practice, this outcome can be obtained by comparing the prediction $\hat{a}$ with the golden answer a based on an LLM.

Problem Definition. Given a failed execution graph $\mathcal{G}$ for question q together with the golden answer a, our objective is to identify the **earliest decisive faulty operation**, denoted as o^*. For an operation o, let $\mathrm{Anc}_{\mathcal{G}}(o)$ and $\mathrm{Desc}_{\mathcal{G}}(o)$ denote its strictly upstream ancestor operations and strictly downstream descendant operations in the execution graph, respectively. An operation o is a can-

didate earliest error if it is faulty and all operations in $\mathrm{Anc}_{\mathcal{G}}(o)$ are functionally correct. To determine whether it is decisive, we construct an idealized execution[1]. Specifically, $\mathcal{G}^{(o,*)}$ replaces the faulty outputs of o with their correct counterparts and assumes ideal execution for every operation in $\mathrm{Desc}_{\mathcal{G}}(o)$. The operation is decisive if this intervention rescues the failed execution, i.e., $Z(\mathcal{G}^{(o,*)}) = 0$. Let $\mathcal{D}(\mathcal{G})$ denote the set of operations satisfying this criterion. Using the execution timestamp t_o only to order decisive candidates, we then define[2]

$$o^* = \arg\min_{o \in \mathcal{D}(\mathcal{G})} t_o. \tag{1}$$

Although our failure attribution objective follows prior work on LLM agent systems (Zhang et al., 2025d;a; Wang et al., 2026), the memory-system setting studied here differs in several important ways. In prior works, the execution trace is often treated as a relatively short sequence of logs produced by a single task run. In contrast, a memory system is executed over a long historical trajectory τ, so its trace can grow to tens of megabytes in our setting. More importantly, the trace is inherently not an unstructured, flat log. For example, a memory unit produced by an earlier memory update may later be retrieved, transformed, or used in answer generation, creating dependencies that span both different operations and different time steps.

3 MemTraceBench Construction

Due to the lack of datasets for evaluating automatic failure attribution in stateful agents with non-parametric memory, we construct a new dataset, `MemTraceBench` (MIT Licence). Figure 5 in Appendix illustrates the overview of construction process. Each example in `MemTraceBench` includes a question, its corresponding golden answer, the full execution trace of the system, and annotated failure information. The annotations include the unique identifiers of faulty operations, their error types, and explanations. We construct our benchmark using question–answer pairs from LoCoMo (Maharana et al., 2024), LongMemEval (Wu et al., 2025), and RealMem (Bian et al., 2026). Four representative memory systems are selected, including long-context memory, RAG (Lewis et al., 2020), Mem0 (Chhikara et al., 2025), and EverMemOS (Hu et al., 2026a). See Appendix D.1 for further details of data sources and memory systems.

Constructing this benchmark requires collecting fine-grained execution graphs for memory systems, rather than only the inputs and outputs of LLM calls. In particular, we need to capture how messages produce memory units, how memories evolve over time, and how intermediate variables depend on one another across memory construction, retrieval, response generation, and evaluation. Since existing memory systems use heterogeneous schemas and code structures, we collect traces through explicit instrumentation rather than rewriting them around a unified abstraction. Moreover, existing instrumentation-based tracing frameworks are mostly event-centric and do not directly track variable evolution and dependencies. We therefore develop `smartcomment`, a lightweight tracing package for recording developer-specified operations, variables, and their dependencies. We instrument each memory system by adding tracing statements at key operations and then run the instrumented systems on sampled trajectories, collecting 1,514 distinct errors across all systems (see Appendix D.2 for more details). **We then recruit five annotators from the author team to identify the faulty operations, and provide corresponding error types and explanations.** The final benchmark contains 160 system-related failure cases. Appendix D.4 shows the annotation process. We also provide more details on `smartcomment` in Appendix E.

[1] The idealized execution is used only to specify what counts as a decisive faulty operation in our problem definition. It is not a replay step performed in practice. Replaying all affected downstream operations is prohibitively expensive, especially for long-horizon interactions, and may not faithfully realize the intended counterfactual because downstream modules can still fail even after the upstream error has been corrected.

[2] Real-world failures may involve multiple independent faulty operations. Our primary focus is the singleton setting, which allows us to isolate the core problem of attributing a failure to its earliest decisive faulty operation. This setting is already challenging in practice and provides a natural foundation for more general joint attribution. We discuss possible extensions of `MemTrace` to the non-singleton setting in Appendix F.

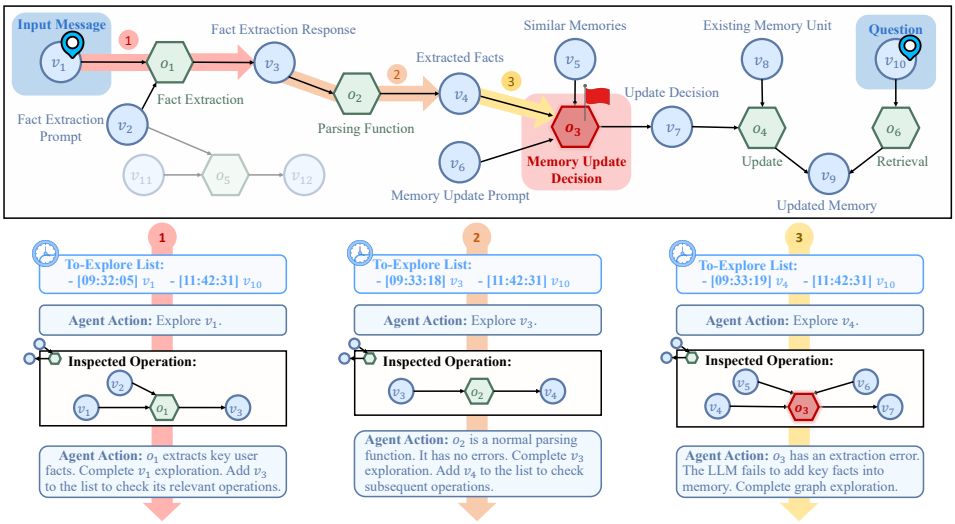


Figure 2: **An illustrative workflow of `MemTrace`.** The initial to-explore list contains v_1 and v_{10}. Starting from v_1, the agent inspects the operation subgraph corresponding to the operation o_1, and finds that o_1 correctly extracts the key user facts. The agent then adds v_3 to the list to inspect subsequent operations. By continuing this graph exploration process, the agent identifies the faulty operation o_3 in the third iteration.

4 METHODOLOGY

We propose `MemTrace` to automatically attribute failures in non-parametric memory systems. It casts failure attribution as an agentic graph exploration problem. As illustrated in Figure 2, the agent iteratively inspects local operation subgraphs in $\mathcal{G}$ and updates its exploration state until it identifies the target decisive error o^* or reaches the maximum number of reasoning steps. At each iteration, `MemTrace` maintains a bounded to-explore list of size at most N. The list is implemented as a priority queue over variable nodes. Each variable $v \in \mathcal{V}$ is associated with its insertion timestamp t_v in the execution graph. Variables with earlier timestamps are assigned higher priority in the list. This priority ensures the agent inspects earlier operations first. The overall method contains three modules: selecting starting points, exploring the execution graph, and managing the agent's working context.

4.1 INITIALIZATION OF STARTING POINT

Before exploring the graph, `MemTrace` needs to choose a small set of starting variables. A naive strategy is to initialize the to-explore list with all system inputs, including the question q and all raw input messages $\{m_i\}_{i=1}^n$ in the historical trajectory τ. However, this creates a very large search space, especially when the trajectory spans many sessions.

To reduce the initial branching factor, `MemTrace` uses hybrid retrieval to identify source messages that are most likely to contain the critical information needed by the failed question. Specifically, we construct a retrieval query by concatenating the question with the golden answer. We then perform both dense retrieval and sparse retrieval over the raw message set $\{m_i\}_{i=1}^n$ to obtain top-N candidate messages from each retriever. The two ranked lists are fused by Reciprocal Rank Fusion (RRF), and the top $\lfloor N/2 \rfloor$ messages from the fused ranking are selected. Finally, these messages together with the question q are used to create the initial to-explore list $\mathcal{L}_0$. Note that we reserve the remaining capacity so that the agent can add newly discovered downstream variables during exploration.

4.2 EXECUTION GRAPH EXPLORATION

At the t-th iteration, given the current to-explore list $\mathcal{L}_{t-1}$, `MemTrace` selects the variable v_t with the earliest timestamp and marks it as the variable under exploration. It then retrieves all operations

directly involving this variable:

$$\mathcal{O}(v_t) = \{o \in \mathcal{O} \mid v_t \in \mathrm{In}(o) \cup \mathrm{Out}(o)\}. \tag{2}$$

For each operation $o \in \mathcal{O}(v_t)$, `MemTrace` converts the corresponding operation-level subgraph $\mathcal{G}_o$ into a textual representation, including the operation name, category, comment, input variables, output variables and dependency relations. This localized view allows the agent to reason over the part of the execution graph that is immediately relevant to the current variable, instead of loading the entire graph into context. The agent judges each inspected operation according to the decisive-error criterion defined in Section 2. If an operation is locally correct, the agent follows the information flow downstream by adding relevant downstream variables of the operation subgraph into the list:

$$\mathcal{L}_t = (\mathcal{L}_{t-1} \setminus \{v_t\}) \cup \mathcal{A}_t, \tag{3}$$

where $\mathcal{A}_t \subseteq \mathcal{V}$ is the set of newly selected variables to explore next. This process encourages `MemTrace` to track the lifetime of critical information through the memory system. The exploration terminates when the agent identifies o^*, or when the maximum number of reasoning iterations is reached.

4.3 Working Context Management

Execution graphs for memory systems can be large, often spanning many operations and long variable values. Therefore, `MemTrace` explicitly manages the agent's working context during graph exploration. From the action space of the agent, `MemTrace` supports a lightweight preview mode for each operation subgraph. In this mode, concrete variable values are omitted. The agent can then selectively inspect only the variables that are relevant to its current hypothesis. For large variable values, `MemTrace` provides targeted access through pagination and regex search. The textual representation of operation subgraphs can also be paginated. These tool-level controls prevent sudden context expansion. In addition, `MemTrace` automatically applies working-context summarization when the context exceeds a predefined safety threshold T.

4.4 Search-Based Operation Exploration

The graph-based exploration strategy in `MemTrace` requires the agent to move between variables by following dependency edges, and at each step the agent can only inspect operations involving the current variable. This design can be inefficient when the execution graph is weakly structured or degenerates into a long chain. To handle such cases, we introduce `MemTrace-OBS`.

`MemTrace-OBS` is based on the observation that operation names, variable values, and comments often already reveal the approximate information flow and functional role of each operation. Concretely, it converts each operation-level subgraph into a textual operation block. In this block, dependency edges and unique variable identifiers are removed, while the input variables, output variables, intermediate variables, and operation attributes such as the operation name and comment are preserved. This compressed representation reduces token usage, especially for operations with many repetitive edges[3]. We then sort all operation blocks by timestamp and concatenate them with separators to form a weakly structured operation log. Inspired by search mechanisms used by coding agents to navigate large codebases (Yang et al., 2024b), `MemTrace-OBS` equips the agent with a global operation-search tool. Given a regular expression, the tool returns operation blocks whose textual contents match the query, with a configurable limit on the maximum number of returned blocks.

5 Experiments

5.1 Experimental Setup

Baselines and Backbones. We study failure attribution for stateful agents with non-parametric memory systems, the problem for which no prior baseline is specifically designed (see related dis-

[3]For example, when retrieving 100 memory units, the query may be connected to every retrieved memory by edges with nearly identical attributes. Representing these edges adds substantial overhead but little additional information.

Backbone	Method	Long-Context		RAG		Mem0		EverMemOS		Overall	
		ETA	OIA	ETA	OIA	ETA	OIA	ETA	OIA	ETA	OIA
GPT-4.1 mini	All-at-Once	30.00	5.00	40.00	27.50	7.50	7.50	50.00	12.50	31.88	13.13
	MemTrace-OBS	9.17	3.33	25.83	17.5	33.33	16.67	11.67	0.0	20.00	9.38
	MemTrace	20.83	4.17	41.67	26.67	35.83	23.33	47.50	2.50	**36.46**	**14.17**
GPT-5.4	All-at-Once	40.00	0.00	80.00	52.50	35.00	5.00	52.50	17.50	51.88	18.75
	MemTrace-OBS	7.50	7.50	87.50	87.50	60.00	55.00	60.00	35.00	53.75	**46.25**
	MemTrace	20.00	20.00	72.50	65.83	70.00	59.17	55.00	7.50	**54.38**	38.13

Table 1: **Failure attribution accuracy on MemTraceBench across memory systems.** "ETA" and "OIA" denote the accuracy of error type prediction and faulty operation identification, respectively. All values are reported as percentages. "Overall" aggregates results across all memory systems.

Backbone	Method	Long-Context		RAG		Mem0		EverMemOS		Overall	
		Tokens	Time	Tokens	Time	Tokens	Time	Tokens	Time	Tokens	Time
GPT-4.1 mini	All-at-Once	1,164.79	0.85	1,199.22	0.85	1,204.58	0.77	1,105.65	0.77	1,168.56	**0.81**
	MemTrace-OBS	692.79	2.62	684.95	1.90	1,077.10	2.11	981.82	3.63	**859.17**	2.57
	MemTrace	4,471.10	7.06	839.48	3.84	830.85	4.56	1,126.10	3.82	1,816.88	4.82
GPT-5.4	All-at-Once	1,150.14	0.83	1,199.35	0.85	1,204.74	1.02	1,120.69	0.87	1,168.73	**0.89**
	MemTrace-OBS	373.89	1.11	277.32	0.93	210.00	0.75	333.67	1.07	**298.72**	0.97
	MemTrace	2,524.81	5.11	1,477.03	3.41	846.09	2.19	2,654.01	5.63	1,875.49	4.09

Table 2: **Average inference cost on MemTraceBench across memory systems.** "Tokens" denotes the average token cost (in thousands) required to run automatic failure attribution for one error case, including both total input and output tokens. "Time" denotes the average end-to-end runtime (in minutes) for one error case.

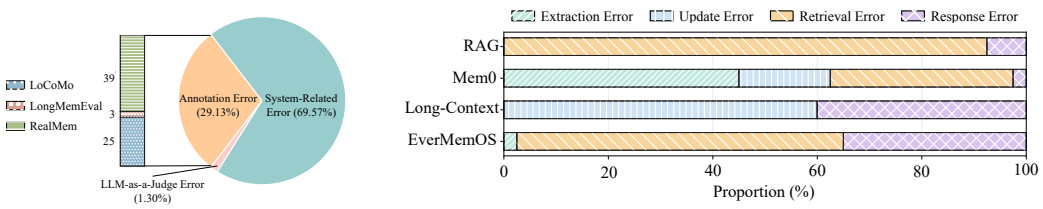
cussion in Appendix B). We therefore consider All-at-Once (Zhang et al., 2025d), a general failure-attribution approach that directly analyzes the flattened log. GPT-4.1 mini (OpenAI, 2025) and GPT-5.4 (OpenAI, 2026) are used as the agent backbones. More details are provided in Appendix G.

Evaluation Metrics. We evaluate failure attribution quality using two metrics. **Error type prediction accuracy** measures whether the agent-predicted error type matches the annotated error type in MemTraceBench. **Faulty operation identification accuracy** measures whether the operation identifier predicted by the agent matches the annotated faulty operation identifier. In addition to attribution accuracy, we report the average token cost and average end-to-end runtime, since practical deployment of automatic failure attribution must handle large volumes of execution logs.

5.2 Main Results

All-at-Once is ill-suited to multi-session stateful memory systems. All-at-Once achieves only 18.75% overall OIA on GPT-5.4, substantially below both MemTrace and MemTrace-OBS. It also underperforms MemTrace with GPT-4.1 mini. Because it discards many early operations, it favors retrieval and response errors near the end of execution. Its performance therefore varies with the error distributions shown in Figure 3b.

Graph-based exploration is particularly effective when errors arise during memory construction or when smaller LLMs are used. As shown in Table 1, MemTrace achieves the highest overall ETA with both backbones, with the clearest improvement under GPT-4.1 mini, where it outperforms MemTrace-OBS from 20.00% to 36.46%. We find that search-based operation exploration exhibits a late-stage inspection bias across both backbones. Given global access to operation blocks, the agent often searches for keywords from the golden answer and directly jumps to operations near retrieval or response. The effect of this behavior depends on model capability. GPT-4.1 mini is more susceptible to hard distractors in the retrieved context. It may incorrectly regard retrieval as successful and infer that the missing evidence is lost during extraction. This causes retrieval and response errors to be misclassified as extraction errors. GPT-5.4 more reliably distinguishes decisive evidence from such distractors. Consequently, direct access to late-stage operations is advantageous on RAG and EverMemOS subsets where retrieval and response errors constitute a large proportion of failures. In contrast, failures in Mem0 and Long-Context more frequently orig-



(a) Error distribution across datasets

(b) System-related error distribution across memory systems

Figure 3: **Overview of error distribution in `MemTraceBench`.**

inate during memory extraction or update. By explicitly following dependency edges from source messages through memory extraction and subsequent updates, `MemTrace` preserves information provenance and is better suited to localizing such early-stage faults.

Search-based operation exploration offers an efficient alternative for weakly structured traces. Table 2 shows that `MemTrace-OBS` consistently incurs the lowest overall inference cost across both backbones. It only uses 15.25% of the tokens and 29.42% of the runtime required by `MemTrace` in average on the long-context subset. This advantage is expected because long-context memory performs repeated context updates, producing traces with weak graph structure. By contrast, on the Mem0 subset, `MemTrace-OBS` uses 76.75% of the tokens and 40.26% of the runtime of `MemTrace`. RAG and EverMemOS show larger cost reductions than Mem0, likely because both systems maintain and update message buffers before triggering indexing or extraction. This makes parts of their execution graphs locally resemble the structure of long-context memory execution graph. Although `MemTrace` remains more expensive than `MemTrace-OBS`, it is still substantially faster and cheaper than manual attribution performed by human experts.

5.3 Analysis

LLM-identified errors are highly reliable. Figure 3 shows that whenever the LLM judge identifies an error, the verdict is almost always correct. Inspecting the small set of disagreements between the LLM judge and our annotators (see Figure 9), we find that the judge is overly strict. It penalizes responses that are essentially correct but either overly verbose or lacking sufficient specificity.

High-quality annotation is intrinsically difficult on long-horizon memory benchmarks. Despite careful human verification, all three datasets contain some annotation errors (see Figures 10 and 11). We find that these errors typically arise from imprecise questions, insufficient source evidence, or inconsistencies between the golden answer and the supporting evidence. In RealMem, the questions and reference answers are substantially more open-ended than those in the other two datasets, which further amplifies annotator subjectivity.

Error distributions reveal distinct bottlenecks across memory systems. As shown in Figure 3b, different memory systems exhibit substantially different failure patterns. RAG has no extraction errors because it does not contain an extraction module, whereas Mem0 and EverMemOS both rely on extraction. Notably, EverMemOS produces very few extraction errors. We find that its extraction module is more robust, more generalizable across both human-assistant and multi-party dialogues, and more effective at handling temporal information. For retrieval errors, Mem0, EverMemOS, and RAG all fail frequently, partly because we retrieve only the top-10 memory units. However, this also indicates existing retrieval modules still struggle to recall all target evidence under a limited retrieval budget. For EverMemOS, some retrieval failures further originate from the final reranker, which fails to rerank target memories into the top-10 candidates. Long-context memory, by design, do not perform retrieval and therefore have no retrieval errors. We also observe no deletion errors, likely because deletion is only supported by Mem0 and is rarely tested in current benchmarks[4]. Compared with other systems, Mem0 additionally supports memory updates, which leads

[4]After analyzing the execution logs of Mem0, we find delete operations account for only 1.02% of all add, update, and delete operations in Mem0.

to more diverse error modes. Finally, all systems exhibit response errors, showing even when related memories are retrieved successfully, effectively using them to give the final answer remains an open challenge.

Source evidence and system prior knowledge improve automatic failure attribution. In realistic development settings, practitioners often have access to two forms of auxiliary information: a high-level understanding of the memory-system pipeline, and source evidence provided by the evaluation set for debugging and iteration. We therefore study whether `MemTrace` can benefit from initializing the to-explore list with source evidence and from adding a coarse pipeline description to the task instruction. This analysis is conducted on a `MemTraceBench` subset that excludes the long-context memory category. As shown in Table 3, source evidence significantly improves OIA as it provides accurate starting points for graph exploration. It also reduces attribution cost, since each question-answer pair usually contains only a small number of golden source-evidence messages, typically one to four. This keeps the initial to-explore list small. Adding prior knowledge about the memory pipeline also improves OIA. However, it increases token cost due to additional system-level prompts. Combining both sources yields the best attribution performance with lower token usage and runtime than the original setting.

Method	ETA	OIA	Tokens	Time
GPT-4.1 mini				
`MemTrace`	41.67	17.50	932.14	4.07
+ Source Evidence	40.55	27.22	575.69	**1.14**
+ Prior Knowledge	**46.39**	23.89	947.32	3.69
+ Both	45.83	**29.44**	**521.80**	1.43
GPT-5.4				
`MemTrace`	65.83	44.17	1,659.04	3.74
+ Source Evidence	69.17	54.17	**1,036.69**	**2.41**
+ Prior Knowledge	64.17	45.83	1,837.41	4.94
+ Both	**70.00**	**58.33**	1,475.29	3.06

Table 3: **Additional analysis of `MemTrace` on the subset of `MemTraceBench`.** "+ Both" indicates adding both source evidence and prior knowledge.

Golden answers provide effective signals for retrieving source evidence. We further examine whether concatenating the original question with the golden answer can effectively retrieve source evidence for initializing graph exploration. We compare three retrieval strategies: sparse retrieval, dense retrieval, and hybrid retrieval used by `MemTrace`. We evaluate retrieval performance using Recall@8. As shown in Table 5a, hybrid retrieval consistently achieves the best performance across all datasets. On LoCoMo and LongMemEval, even the weakest method achieves over 70%. On RealMem, dense retrieval underperforms sparse retrieval, possibly because its messages are longer, a setting in which sparse retrieval can be more reliable (Luan et al., 2021). Overall, these results demonstrate golden answers can provide high-quality starting points for `MemTrace` exploration. Appendix J.1 further analyzes source-evidence retrieval when golden answers are unavailable.

6 Application

6.1 Diagnostic Report for Memory Systems

Analyzing memory-system failures is labor-intensive due to long construction, update, retrieval, and response pipelines. MemTrace enables operation-level aggregation of failures, automatically summarizing where and how systems fail across the pipeline. We apply this analysis to Mem0 and EverMemOS. Tables 6–8 and Tables 9–10 present the generated reports for Mem0 and EverMemOS, respectively. The report generation details are presented in Appendix H. The generated reports reveal different failure patterns in Mem0 and EverMemOS. For Mem0, the extraction module tends to keep high-level user information while dropping fine-grained details, consistent with Hu et al. (2026b). The report further identifies timestamp reassignment during updates, where content remains unchanged but its time is modified. For EverMemOS, no major extraction errors are observed, but aggregation and counting failures appear in the response stage. It also localizes issues to specific retrieval components, including the reranker, sufficiency checker, and query reformulation module. **Overall, `MemTrace` reliably captures overall failure patterns and enables fine-grained diagnosis of concrete pipeline components.**

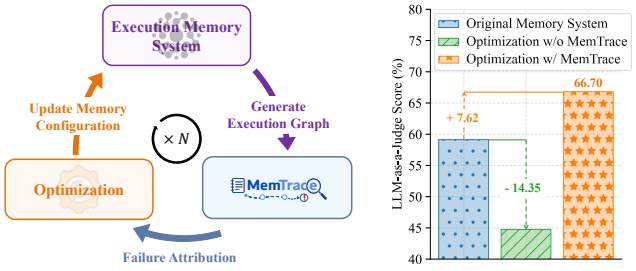
6.2 Automatic Optimization of Prompts in Memory Systems

Non-parametric memory systems rely on many hand-written prompts, making manual tuning costly. A natural approach is automatic prompt optimization, but existing methods fail in multi-session settings due to long, cross-session execution traces (see Appendix I.1). We decouple *credit assignment* from *prompt rewriting* with `smartcomment` and `MemTrace`. As illustrated in Figure 4a, `smartcomment` records the runtime execution graph of the memory system, and `MemTrace` performs credit assignment on this graph to localize the earliest decisive faulty operation. **Once that operation is identified, prompt optimization reduces to a local problem: we only need to invoke an off-the-shelf optimizer on the small set of prompts participating in that operation.** This sidesteps the difficulties of prior approaches simultaneously, since we never have to fit the full trajectory into the optimizer's context, propagate textual signals along a long causal chain, or replay the entire memory pipeline.

We evaluate this closed-loop pipeline on Mem0 and LoCoMo using an LLM-as-a-judge score. To isolate the effect of `MemTrace`, we introduce a no-attribution baseline. As shown in Figure 4b, `MemTrace`-guided optimization improves performance from 59.08% to 66.70% (+7.62 points), whereas the no-attribution baseline decreases it to 44.73%. **These results demonstrate the importance of localized credit assignment: by identifying the operations responsible for failures, `MemTrace` provides meaningful insights that enable the optimizer to make effective updates, even though `MemTrace` itself is imperfect.** Further experimental details and cost analysis are provided in Appendix I.2.

7 Related Work

Recent LLM memory systems aim to support long-horizon interactions by dynamically extracting, updating, forgetting, and maintaining memories across sessions (Packer et al., 2023; Zhong et al., 2024; Xu et al., 2025; Chhikara et al., 2025; Cao et al., 2025; Wang & Chen, 2025; Fang et al., 2025; Liu et al., 2026; Hu et al., 2026a; Yang et al., 2026). These systems introduce complex execution pipelines, making failures difficult to localize and attribute. Existing work on diagnosing LLM agents mainly focuses on identifying faulty steps in short reasoning traces within a single task instance, using sampling-based signals, process-level supervision, or LLM-based inspection of intermediate trajectories (Xiong et al., 2024; Lightman et al., 2024; Wang et al., 2025a; Zhang et al., 2025a; Ge et al., 2025; Baker et al., 2025; Zhang et al., 2025d; Yüksekgönül et al., 2025; Lee et al., 2026; Li et al., 2026; Wang et al., 2026). In contrast, failures in stateful agents with long-term memory may originate from earlier sessions and must be distinguished from substantial irrelevant interaction history. For more related work, see Appendix A.



(a) Pipeline (b) Performance

Figure 4: **Automatic optimization of Mem0.** (a) Overview of the optimization pipeline. (b) Performance comparison of Mem0 before and after optimization, showing an improvement of 7.62 percentage points after three rounds.

8 Conclusion

In this work, we study a new research question: how to automatically diagnose failures in non-parametric memory systems. To this end, we construct `MemTraceBench`, a benchmark built from publicly available datasets and open-source memory systems. Based on this benchmark, we propose `MemTrace` that attributes memory-system failures to concrete operations in the execution pipeline. Our findings show that precisely localizing failures remains challenging. We hope our work will encourage further research on automatic failure diagnosis for memory-augmented AI systems.

AI USE STATEMENT

Generative AI is used both as an experimental component and as a limited research and writing assistant. As detailed in the paper, LLMs are used throughout the experimental pipeline for memory system execution, failure attribution, evaluation, prompt optimization, and diagnostic report generation. MemTraceBench builds on public datasets containing LLM-synthesized interaction trajectories. During benchmark construction, three human annotators independently provide labels and explanations for each failure case. After the final error type and faulty operation have been determined by the annotators, an LLM is only used to aggregate and polish their explanations. Generative AI tools are additionally used for minor code completion, editing, debugging, and language polishing of the manuscript. They are not used to formulate the research question, design the methodology, conduct the literature review, or determine the final conclusions. All AI-assisted text, code, annotations, and other artifacts included in this work are manually reviewed. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

`MemTrace` is intended to support failure diagnosis and transparency for non-parametric memory systems. In this work, `MemTraceBench` is constructed from publicly available, fully LLM-synthesized user trajectories and does not contain real user interactions. However, applying `MemTrace` to real-world memory systems may involve execution traces that contain sensitive user information (Chen et al., 2026). Such use requires careful data governance, including informed consent, access control, anonymization when possible, and secure storage of logs and generated reports. It is also critical to acknowledge that the diagnoses produced by `MemTrace` may be imperfect. Consequently, they should be treated as assistive evidence rather than definitive judgments, with human review required before drawing reliability conclusions or deploying system changes.

REPRODUCIBILITY STATEMENT

All datasets and models used in this work are publicly accessible through their respective official sources or APIs. Details of the data construction process are provided in Appendix D, and Appendix E.3 presents a concrete example of inserting `smartcomment` statements into the source code of a memory system. The main text provides the key implementation details of our experiments. Due to space constraints, additional implementation details, including experimental configurations and procedures, are provided in Appendices G, H, and I.2. Upon acceptance, we will release the dataset, source code, annotation website, and prompts used in this work on GitHub. These resources will enable researchers to reproduce our experimental results and facilitate further research on automatically identifying and diagnosing errors in memory systems.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe (eds.), *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei A. Zaharia, and O. Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. *ArXiv*, abs/2507.19457, 2025. URL https://api.semanticscholar.org/CorpusID:280046245.

Anthropic. Introducing claude opus 4.5, 2025. URL https://www.anthropic.com/news/claude-opus-4-5.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=hSyW5go0v8.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Mądry, Wojciech Zaremba, Jakub W. Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *ArXiv*, abs/2503.11926, 2025. URL https://api.semanticscholar.org/CorpusID:276937204.

Yuanchen Bei, Tianxin Wei, Xuying Ning, Yanjun Zhao, Zhining Liu, Xiao Lin, Yada Zhu, Hendrik Hamann, Jingrui He, and Hanghang Tong. Mem-gallery: Benchmarking multimodal long-term conversational memory for mllm agents. *ArXiv*, abs/2601.03515, 2026. URL https://api.semanticscholar.org/CorpusID:284532253.

Haonan Bian, Zhiyuan Yao, Sen Hu, Zishan Xu, Shaolei Zhang, Yifu Guo, Ziliang Yang, Xueran Han, Huacan Wang, and Ronghao Chen. Realmem: Benchmarking llms in real-world memory-driven interaction. *ArXiv*, abs/2601.06966, 2026. URL https://api.semanticscholar.org/CorpusID:284647968.

Akash Bonagiri, Devang Borkar, Gerard Janno Anderias, Setareh Rafatirad, and Houman Homayoun. Causalflow: Causal attribution and counterfactual repair for llm agent failures. *ArXiv*, abs/2605.25338, 2026. URL https://api.semanticscholar.org/CorpusID:288671092.

Zouying Cao, Jiaji Deng, Li Yu, Wei Zhou, Zhaoyang Liu, Bolin Ding, and Haiquan Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. *ArXiv*, abs/2512.10696, 2025. URL https://api.semanticscholar.org/CorpusID:283737683.

Ding Chen, Simin Niu, Kehan Li, Peng Liu, Xiang Zheng, Bo Tang, Xinchi Li, Feiyu Xiong, and Zhiyu Li. Halumem: Evaluating hallucinations in memory systems of agents. *ArXiv*, abs/2511.03506, 2025. URL https://api.semanticscholar.org/CorpusID:282758378.

Yining Chen, Jihao Zhao, Bo Tang, Hongya Wang, Yue Zhang, Fei Huang, Feiyu Xiong, and Zhiyu Li. Memprivacy: Privacy-preserving personalized memory management for edge-cloud agents. 2026. URL https://api.semanticscholar.org/CorpusID:288253755.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/83ba7056bce2c3c3c27e17397cf3e1f0-Abstract-Conference.html.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. In *European Conference on Artificial Intelligence*, 2025. URL https://api.semanticscholar.org/CorpusID:278165315.

Yiming Du, Bingbing Wang, Yangfan He, Bin Liang, Baojun Wang, Zhongyang Li, Lin Gui, Jeff Z. Pan, Ruifeng Xu, and Kam-Fai Wong. Memguide: Intent-driven memory selection for goal-oriented multi-session llm agents. In *AAAI Conference on Artificial Intelligence*, 2025. URL https://api.semanticscholar.org/CorpusID:278911777.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva N. Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *ArXiv*, abs/2404.16130, 2024. URL https://api.semanticscholar.org/CorpusID:269363075.

Shahul ES, Jithin James, Luis Espinosa Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval augmented generation. In Nikolaos Aletras and Orphée De Clercq (eds.), *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2024 - System Demonstrations, St. Julians, Malta, March 17-22, 2024*, pp. 150–158. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.EACL-DEMO.16. URL https://doi.org/10.18653/v1/2024.eacl-demo.16.

Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Huajun Chen, and Ningyu Zhang. Lightmem: Lightweight and efficient memory-augmented generation. *ArXiv*, abs/2510.18866, 2025. URL https://api.semanticscholar.org/CorpusID:282245881.

Tian Gao. VizTracer - A debugging and profiling tool that can trace and visualize python code execution, 2020. URL https://github.com/gaogaotiantian/viztracer.

Yu Ge, Linna Xie, Zhong Li, Yu Pei, and Tian Zhang. Who is introducing the failure? automatically attributing failures of multi-agent systems via spectrum analysis. *ArXiv*, abs/2509.13782, 2025. URL https://api.semanticscholar.org/CorpusID:281332885.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Yuanzhuo Wang, and Jian Guo. A survey on llm-as-a-judge. *ArXiv*, abs/2411.15594, 2024. URL https://api.semanticscholar.org/CorpusID:274234014.

Bernal Jimenez Gutierrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/6ddc001d07ca4f319af96a3024f6dbd1-Abstract-Conference.html.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Zeming Chen, Tong Wu, Siru Ouyang, Zihan Wang, Jiaxin Pei, Julian McAuley, Yejin Choi, and Alex 'Sandy' Pentland. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *ArXiv*, abs/2602.16313, 2026. URL https://api.semanticscholar.org/CorpusID:285726166.

Chuanrui Hu, Xingze Gao, Zuyi Zhou, Dannong Xu, Yi Bai, Xintong Li, Hui Zhang, Tong Li, Chong Zhang, Lidong Bing, and Yafeng Deng. Evermemos: A self-organizing memory operating system for structured long-horizon reasoning. *ArXiv*, abs/2601.02163, 2026a. URL https://api.semanticscholar.org/CorpusID:284488385.

Sen Hu, Zhiyu Zhang, Yuxiang Wei, Xueran Han, Zhenheng Tang, Huacan Wang, and Ronghao Chen. Clonemem: Benchmarking long-term memory for ai clones. *ArXiv*, abs/2601.07023, 2026b. URL https://api.semanticscholar.org/CorpusID:284648447.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in llm agents via incremental multi-turn interactions. *ArXiv*, abs/2507.05257, 2025. URL https://api.semanticscholar.org/CorpusID:280136659.

Zhanghao Hu, Qinglin Zhu, Hanqi Yan, Yulan He, and Lin Gui. Beyond rag for agent memory: Retrieval by decoupling and aggregation. *ArXiv*, abs/2602.02007, 2026c. URL https://api.semanticscholar.org/CorpusID:285269925.

Suizhi Huang, Mei Li, Han Yu, and Xiaoxiao Li. Textresnet: Decoupling and routing optimization signals in compound ai systems via deep residual tuning. *ArXiv*, abs/2602.08306, 2026. URL https://api.semanticscholar.org/CorpusID:285451612.

Bowen Jiang, Yuan Yuan, Maohao Shen, Zhuoqun Hao, Zhangchen Xu, Zichen Chen, Ziyi Liu, Anvesh Rao Vijjini, Jiashu He, Hanchao Yu, Radha Poovendran, Greg Wornell, Lyle Ungar, Dan Roth, Sihao Chen, and Camillo Jose Taylor. Personamem-v2: Towards personalized intelligence via learning implicit user personas and agentic memory. *ArXiv*, abs/2512.06688, 2025. URL https://api.semanticscholar.org/CorpusID:283693901.

Peiling Jiang, Fuling Sun, and Haijun Xia. Log-it: Supporting programming with interactive, contextual, structured, and visual logs. In Albrecht Schmidt, Kaisa Väänänen, Tesh Goyal, Per Ola Kristensson, Anicia Peters, Stefanie Mueller, Julie R. Williamson, and Max L. Wilson (eds.), *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*, pp. 594:1–594:16. ACM, 2023. doi: 10.1145/3544548.3581403. URL https://doi.org/10.1145/3544548.3581403.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. Dspy: compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2023.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation. *ArXiv*, abs/2505.06120, 2025. URL https://api.semanticscholar.org/CorpusID:278481320.

Langfuse Team. Langfuse, 2023. URL https://github.com/langfuse/langfuse.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, et al. Codetracer: Towards traceable agent states. *arXiv preprint arXiv:2604.11641*, 2026.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Dongmei Jiang, and Liqiang Nie. Optimus-1: Hybrid multimodal memory empowered agents excel in long-horizon tasks. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024a. URL http://papers.nips.cc/paper_files/paper/2024/hash/5949a8750a110ce1f0631b1776c500a2-Abstract-Conference.html.

Zhiyu Li, Shichao Song, Hanyu Wang, Simin Niu, Ding Chen, Jiawei Yang, Chenyang Xi, Huayi Lai, Jihao Zhao, Yezhaohui Wang, Junpeng Ren, Zehao Lin, Jiahao Huo, Tianyi Chen, Kai Chen, Ke-Rong Li, Zhiqiang Yin, Qingchen Yu, Bo Tang, Hongkang Yang, Zhiyang Xu, and Feiyu Xiong. Memos: An operating system for memory-augmented generation (mag) in large language models. *ArXiv*, abs/2505.22101, 2025. URL https://api.semanticscholar.org/CorpusID:278960153.

Zhuowan Li, Cheng Li, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. Retrieval augmented generation or long-context llms? A comprehensive study and hybrid approach. In Franck Dernoncourt, Daniel Preotiuc-Pietro, and Anastasia Shimorina (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: EMNLP 2024 - Industry Track, Miami, Florida, USA, November 12-16, 2024*, pp. 881–893. Association for Computational Linguistics, 2024b. doi: 10.18653/V1/2024.EMNLP-INDUSTRY.66. URL https://doi.org/10.18653/v1/2024.emnlp-industry.66.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Simplemem: Efficient lifelong memory for llm agents. *ArXiv*, abs/2601.02553, 2026. URL https://api.semanticscholar.org/CorpusID:284512931.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Trans. Assoc. Comput. Linguistics*, 12:157–173, 2024a. doi: 10.1162/TACL_A_00638. URL https://doi.org/10.1162/tacl_a_00638.

Yiqi Liu, Nafise Sadat Moosavi, and Chenghua Lin. Llms as narcissistic evaluators: When ego inflates evaluation scores. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, Findings of ACL, pp. 12688–12701. Association for Computational Linguistics, 2024b. doi: 10.18653/V1/2024.FINDINGS-ACL.753. URL https://doi.org/10.18653/v1/2024.findings-acl.753.

Yi Luan, Jacob Eisenstein, Kristina Toutanova, and Michael Collins. Sparse, dense, and attentional representations for text retrieval. *Trans. Assoc. Comput. Linguistics*, 9:329–345, 2021. doi: 10.1162/TACL_A_00369. URL https://doi.org/10.1162/tacl_a_00369.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 13851–13870. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.747. URL https://doi.org/10.18653/v1/2024.acl-long.747.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/6f1d43d5a82a37e89b0665b33bf3a182-Abstract-Conference.html.

Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, Seunghyun Yoon, and Hinrich Schütze. Nolima: Long-context evaluation beyond literal matching. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/modarressi25a.html.

OpenAI. Introducing gpt-4.1 in the api, 2025. URL https://openai.com/index/gpt-4-1/.

OpenAI. Introducing gpt-5.4, 2026. URL https://openai.com/index/introducing-gpt-5-4/.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pp. 9340–9366. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.EMNLP-MAIN.525. URL https://doi.org/10.18653/v1/2024.emnlp-main.525.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph Gonzalez. Memgpt: Towards llms as operating systems. *ArXiv*, abs/2310.08560, 2023. URL https://api.semanticscholar.org/CorpusID:263909014.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In Sean Follmer, Jeff Han, Jürgen Steimle, and Nathalie Henry Riche (eds.), *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023- 1 November 2023*, pp. 2:1–2:22. ACM, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In Dawn Song, Michael Carbin, and Tianqi Chen (eds.), *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4-8, 2023*. mlsys.org, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/hash/c4be71ab8d24cdfb45e3d06dbfca2780-Abstract-mlsys2023.html.

Chen Qian, Peng Wang, Dongrui Liu, Junyao Yang, Dadi Guo, Ling Tang, Jilin Mei, Qihan Ren, Shuai Shao, Yong Liu, Jie Fu, Jing Shao, and Xia Hu. The why behind the action: Unveiling internal drivers via agentic attribution. *ArXiv*, abs/2601.15075, 2026. URL https://api.semanticscholar.org/CorpusID:284917611.

Ram Rachum, Alex Hall, Iori Yanokura, et al. Pysnooper: Never use print for debugging again, jun 2019. URL https://github.com/cool-RR/PySnooper.

Dongyu Ru, Lin Qiu, Xiangkun Hu, Tianhang Zhang, Peng Shi, Shuaichen Chang, Cheng Jiayang, Cunxiang Wang, Shichao Sun, Huanyu Li, Zizhao Zhang, Binjie Wang, Jiarong Jiang, Tong He, Zhiguo Wang, Pengfei Liu, Yue Zhang, and Zheng Zhang. Ragchecker: A fine-grained framework for diagnosing retrieval-augmented generation. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/27245589131d17368cccdfa990cbf16e-Abstract-Datasets_and_Benchmarks_Track.html.

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. RAPTOR: recursive abstractive processing for tree-organized retrieval. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=GN921JHCRw.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, Proceedings of Machine Learning Research, pp. 3319–3328. PMLR, 2017. URL http://proceedings.mlr.press/v70/sundararajan17a.html.

Haoran Tan, Zeyu Zhang, Chen Ma, Xu Chen, Quanyu Dai, and Zhenhua Dong. Membench: Towards more comprehensive evaluation on the memory of llm-based agents. In Wanxiang Che,

Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Findings of ACL, pp. 19336–19352. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.findings-acl.989/.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pp. 10014–10037. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.ACL-LONG.557. URL https://doi.org/10.18653/v1/2023.acl-long.557.

Hanlin Wang, Jian Wang, Chak Tou Leong, and Wenjie Li. Steca: Step-level trajectory calibration for LLM agent learning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Findings of ACL, pp. 11597–11614. Association for Computational Linguistics, 2025a. URL https://aclanthology.org/2025.findings-acl.604/.

Haoming Wang, Haoyang Zou, Huatong Song, Jiazhan Feng, Junjie Fang, Junting Lu, Longxiang Liu, Qinyu Luo, Shihao Liang, Shijue Huang, Wanjun Zhong, Yining Ye, Yujia Qin, Yuwen Xiong, Yuxin Song, Zhiyong Wu, Bo Li, Chen Dun, Chong Liu, Fuxing Leng, Han rui Wang, Hao Yu, Haobin Chen, Hongyi Guo, Jing Su, Jingjia Huang, Kai Shen, Kaiyu Shi, Lin Yan, Pei-Xiong Zhao, Pengfei Liu, Qinghao Ye, Renjie Zheng, Wayne Xin Zhao, Wen Heng, Wenhao Huang, Wenqian Wang, Xiao jun Qin, Yi Lin, Youbing Wu, Zehui Chen, Zihao Wang, Baoquan Zhong, Xinchun Zhang, Xujing Li, YuanFang Li, Zhongkai Zhao, Chengquan Jiang, Faming Wu, Hao Zhou, Jinlin Pang, Li Han, Qianli Ma, Siyao Liu, Songhua Cai, Wenqi Fu, Xin Liu, Zhi Zhang, Bo Zhou, Guoliang Li, Jiajun Shi, Jiale Yang, Jie Tang, Li Li, Taoran Lu, Woyu Lin, Xiao Tong, Xinyao Li, Yichi Zhang, Yu Miao, Zheng-Wang Jiang, Zili Li, Zi-Hao Zhao, Chenxi Li, Dehua Ma, Feng Lin, Ge Zhang, Haihua Yang, Hangyu Guo, Hongda Zhu, Jiaheng Liu, Jun-Yan Du, Kai Cai, Kuanye Li, Lichen Yuan, Mei Han, Minchao Wang, Shuyu Guo, Tianhao Cheng, Xiaobo Ma, Xiao Xiao, Xiaolong Huang, Xinjie Chen, Yi-Zhen Du, Yilin Chen, Yiwen Wang, Zhaojian Li, Zhen Yang, Zhiyuan Zeng, Chaolin Jin, Chen Li, Haolin Chen, Haolin Chen, Jian Chen, Qinghao Zhao, and Guang Shi. Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. *ArXiv*, abs/2509.02544, 2025b. URL https://api.semanticscholar.org/CorpusID:281079943.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025c. URL https://openreview.net/forum?id=OJd3ayDDoF.

Yawen Wang, Wenjie Wu, Junjie Wang, and Qing Wang. From flat logs to causal graphs: Hierarchical failure attribution for llm-based multi-agent systems. *ArXiv*, abs/2602.23701, 2026. URL https://api.semanticscholar.org/CorpusID:286170884.

Yu Wang and Xi Chen. Mirix: Multi-agent memory system for llm-based agents. *ArXiv*, abs/2507.07957, 2025. URL https://api.semanticscholar.org/CorpusID:280277519.

Zichen Wen, Yiyu Wang, Chenfei Liao, Boxue Yang, Junxian Li, Weifeng Liu, Haocong He, Bo-Han Feng, Xuyang Liu, Yuanhuiyi Lyu, Xu Zheng, Xuming Hu, and Linfeng Zhang. Ai for service: Proactive assistance with ai glasses. *ArXiv*, abs/2510.14359, 2025. URL https://api.semanticscholar.org/CorpusID:282139278.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=pZiyCaVuti.

Ruibin Xiong, Yimeng Chen, Dmitrii Khizbullin, Mingchen Zhuge, and Jürgen Schmidhuber. Beyond outlining: Heterogeneous recursive planning for adaptive long-form writing with language models. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, pp. 24678–24714. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.EMNLP-MAIN.1254. URL https://doi.org/10.18653/v1/2025.emnlp-main.1254.

Weimin Xiong, Yifan Song, Xiutian Zhao, Wenhao Wu, Xun Wang, Ke Wang, Cheng Li, Wei Peng, and Sujian Li. Watch every step! LLM agent learning via iterative step-level process refinement. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pp. 1556–1572. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.EMNLP-MAIN.93. URL https://doi.org/10.18653/v1/2024.emnlp-main.93.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *ArXiv*, abs/2502.12110, 2025. URL https://api.semanticscholar.org/CorpusID:276421617.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024a. URL https://openreview.net/forum?id=Bb4VGOWELI.

Jingkang Yang, Shuai Liu, Hongming Guo, Yuhao Dong, Xiamengwei Zhang, Sicheng Zhang, Pengyun Wang, Zitang Zhou, Binzhu Xie, Ziyue Wang, Bei Ouyang, Zhengyu Lin, Marco Cominelli, Zhongang Cai, Bo Li, Yuanhan Zhang, Peiyuan Zhang, Fangzhou Hong, Joerg Widmer, Francesco Gringoli, Lei Yang, and Ziwei Liu. Egolife: Towards egocentric life assistant. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 28885–28900. Computer Vision Foundation / IEEE, 2025. doi: 10.1109/CVPR52734.2025.02690. URL https://openaccess.thecvf.com/content/CVPR2025/html/Yang_EgoLife_Towards_Egocentric_Life_Assistant_CVPR_2025_paper.html.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024b. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Ke Yang, Zixiang Chen, Xuan He, Jize Jiang, Michel Galley, Chenglong Wang, Jianfeng Gao, Jiawei Han, and Cheng Xiang Zhai. Plugmem: A task-agnostic plugin memory module for llm agents. 2026. URL https://api.semanticscholar.org/CorpusID:286238947.

Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. *Proceedings of the Web Conference 2021*, 2021. URL https://api.semanticscholar.org/CorpusID:235324846.

Hongli Yu, Ting Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. Memagent: Reshaping long-context llm with multi-conv rl-based memory agent. *ArXiv*, abs/2507.02259, 2025. URL https://api.semanticscholar.org/CorpusID:280047896.

Xiang Yue, Yuansheng Ni, Tianyu Zheng, Kai Zhang, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, Huan Sun, Yu Su, and

Wenhu Chen. MMMU: A massive multi-discipline multimodal understanding and reasoning benchmark for expert AGI. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*, pp. 9556–9567. IEEE, 2024. doi: 10.1109/CVPR52733.2024.00913. URL https://doi.org/10.1109/CVPR52733.2024.00913.

Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nat.*, 639(8055):609–616, 2025. doi: 10.1038/S41586-025-08661-4. URL https://doi.org/10.1038/s41586-025-08661-4.

Matei A. Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018. URL https://api.semanticscholar.org/CorpusID:83459546.

Andreas Zeller. Yesterday, my program worked. today, it does not. why? In Oscar Nierstrasz and Michel Lemoine (eds.), *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, Lecture Notes in Computer Science, pp. 253–267. Springer, 1999. doi: 10.1007/3-540-48166-4_16. URL https://doi.org/10.1007/3-540-48166-4_16.

Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pp. 1–10. ACM, 2002. doi: 10.1145/587051.587053. URL https://doi.org/10.1145/587051.587053.

Gui-Min Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. Agentracer: Who is inducing failure in the llm agentic systems? *ArXiv*, abs/2509.03312, 2025a. URL https://api.semanticscholar.org/CorpusID:281091991.

Heng Zhang, Yuling Shi, Xiaodong Gu, Haochen You, Zijian Zhang, Lubin Gan, Yilei Yuan, and Jin Huang. Graphtracer: Graph-guided failure tracing in llm agents for robust multi-turn deep search. *ArXiv*, abs/2510.10581, 2025b. URL https://api.semanticscholar.org/CorpusID:282057258.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Reddy Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *ArXiv*, abs/2510.04618, 2025c. URL https://api.semanticscholar.org/CorpusID:281842432.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025d. URL https://proceedings.mlr.press/v267/zhang25cq.html.

Songhan Zhang, Aoyang Fang, Yifan Yang, Ruiyi Cheng, Xiaoying Tang, and Pinjia He. Dynacausal: Dynamic causality-aware root cause analysis for distributed microservices. *ArXiv*, abs/2510.22613, 2025e. URL https://api.semanticscholar.org/CorpusID:282388804.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *ArXiv*, abs/2506.05176, 2025f. URL https://api.semanticscholar.org/CorpusID:279243736.

Siyan Zhao, Mingyi Hong, Yang Liu, Devamanyu Hazarika, and Kaixiang Lin. Do llms recognize your preferences? evaluating personalized preference following in llms. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=QWunLKbBGF.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pp. 19724–19731. AAAI Press, 2024. doi: 10.1609/AAAI.V38I17.29946. URL https://doi.org/10.1609/aaai.v38i17.29946.

A More Related Work

Non-Parametric Memory Systems for LLMs. LLMs natively rely on the context window as a transient memory buffer, while KV caching (Pope et al., 2023) avoids recomputing repeated prefixes. However, long-context prompting is fundamentally bounded by context length, computational cost, and well-known degradation phenomena such as lost-in-the-middle (Liu et al., 2024a) and failures in long multi-turn conversation (Laban et al., 2025). Retrieval-Augmented Generation (RAG) externalizes memory into non-parametric stores, typically by chunking raw content and retrieving relevant units at inference time (Lewis et al., 2020; Trivedi et al., 2023; Asai et al., 2024; Li et al., 2024b). More recent RAG methods further improve retrieval by refining or summarizing raw content with LLMs before indexing (Sarthi et al., 2024; Gutierrez et al., 2024; Edge et al., 2024). Beyond long-context prompting and RAG, recent memory systems focus on dynamically managing memories during open-ended interactions, including memory extraction (Li et al., 2025; Fang et al., 2025; Hu et al., 2026a; Liu et al., 2026), updating (Xu et al., 2025; Chhikara et al., 2025; Hu et al., 2026c), forgetting (Zhong et al., 2024; Cao et al., 2025), and multi-type memory maintenance (Packer et al., 2023; Wang & Chen, 2025; Yang et al., 2026). Compared with the previous two paradigms, these systems involve substantially more complex execution pipelines, making their failures harder to localize and attribute.

Automatic Failure Attribution. Automatic failure attribution is studied in many domains, including software debugging (Zeller, 1999; 2002), cloud-service diagnosis (Yu et al., 2021; Zhang et al., 2025e), and deep learning analysis (Sundararajan et al., 2017; Meng et al., 2022). The most related line of work focuses on localizing failures in LLM agent systems. Some methods identify faulty steps through sampling-based or process-level signals (Wang et al., 2025a; Xiong et al., 2024; Lightman et al., 2024; Zhang et al., 2025a; Ge et al., 2025). Others use another LLM or agent to inspect intermediate traces and diagnose error locations (Baker et al., 2025; Zhang et al., 2025d). Some approaches further exploit structured trajectories such as trees (Lee et al., 2026; Li et al., 2026) or graphs (Yüksekgönül et al., 2025; Wang et al., 2026) for failure localization. Qian et al. (2026) study failure attribution in memory-driven interaction scenarios, analyzing how retrieved memories influence an agent's downstream actions. This corresponds most closely to response-stage attribution in our work. Overall, prior work mainly targets short reasoning traces within a single task instance. By contrast, in stateful agents with non-parametric memory, failures may be introduced long before the final answer is produced, potentially in earlier sessions, and must be distinguished from substantial irrelevant interaction history.

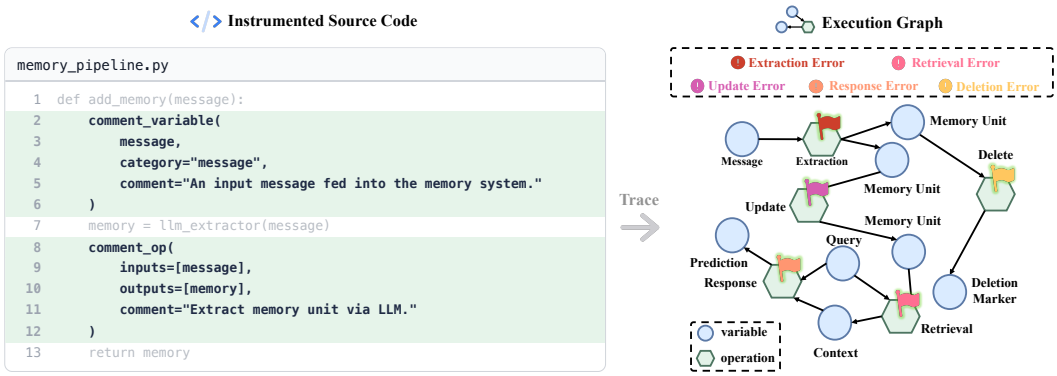


Figure 5: **The overview of dataset construction process.** We define seven error types, five of which are specific to memory systems. We insert `smartcomment`-related code into each memory system and run these systems on public datasets to collect execution graphs and failed cases. For each failed case, once our annotators confirm that it is not caused by annotation errors or LLM-as-a-Judge errors, they identify the earliest decisive faulty operation in the execution graph and provide its cause and error type.

Memory Benchmarks. As memory systems develop rapidly, automatic evaluation for memory systems becomes increasingly important. LoCoMo (Maharana et al., 2024) is one of the earliest and most widely used benchmarks for evaluating long-term memory in LLMs. Later work improves the difficulty of user trajectories (Wu et al., 2025; Bian et al., 2026), increases their diversity (Hu et al., 2025; Tan et al., 2025), or enriches them with multimodal information (Wang & Chen, 2025; Jiang et al., 2025; Bei et al., 2026). Other benchmarks shift the evaluation focus to different scenarios (Zhao et al., 2025; Du et al., 2025; Hu et al., 2026b; He et al., 2026). These benchmarks usually construct question-answer pairs to measure the final performance of memory systems. However, this evaluation paradigm provides limited fine-grained diagnostic information. HaluMem (Chen et al., 2025) offers a more fine-grained automatic evaluation by assessing the accuracy of memory extraction and memory updating. However, it mainly checks whether target memories can be found in the current memory store through retrieval, which may not always reflect the true system behavior. It also cannot reveal when an error is introduced or which operation causes it. In contrast, our work only requires the golden answer, rather than manually provided source evidence or golden memories, to automatically perform fine-grained diagnosis of memory-system failures. Therefore, it can serve as diagnostic infrastructure for different memory benchmarks.

B Detailed Discussion of Prior Failure Attribution Work

`MemTrace` is not the first approach to use execution traces for failure attribution. Its contribution is to study this paradigm in stateful agents with non-parametric memory, where persistent states evolve across sessions and a failure may originate from a much earlier operation relevant to memory construction. Below, we clarify how this setting relates to three groups of closely related work and why they do not provide directly applicable baselines.

Agent Failure Attribution. Prior works on agent failure attribution, such as AgenTracer (Zhang et al., 2025a), CausalFlow (Bonagiri et al., 2026), and GraphTracer (Zhang et al., 2025b), primarily study task-scoped agent trajectories. These methods use counterfactual intervention or replay to identify failure-inducing steps. **Directly applying this strategy to long-term memory systems is computationally difficult**: repairing an early memory operation requires reconstructing the memory state and replaying potentially hundreds of subsequent interactions. Repeating this intervention for every candidate operation would be prohibitively expensive. Moreover, **replay does not perfectly instantiate the idealized intervention in our problem definition**, because errors in later operations may still prevent the final answer from being corrected after an early fault is repaired.

GraphTracer is particularly close to our work because it also represents execution traces as dependency graphs. However, it constructs these graphs by incrementally converting unstructured agent

trajectories into structured dependencies using LLMs and pattern matching, followed by consistency checking. **This construction becomes expensive for long trajectories containing many upstream operations or agents, and dependencies inferred from textual traces are not guaranteed to reproduce the program's actual runtime information flow.** In contrast, `smartcomment` uses lightweight source-code instrumentation to record operation-variable dependencies during execution. It directly captures the lifecycle of persistent memory variables, including creation, in-place update, deletion, retrieval, and downstream use, without reconstructing dependencies from unstructured logs. Adapting GraphTracer to our setting would require both representing versioned memory states and scaling graph construction to multi-session traces containing millions of tokens. A promising future direction is to use LLMs to insert lightweight instrumentation automatically and then obtain runtime-faithful execution graphs by running the instrumented programs.

RAG Diagnostics. Prior works on RAG diagnostics, such as RAGChecker (Ru et al., 2024) and RAGAs (ES et al., 2024), evaluate retrieval and generation quality through component-level metrics computed from the final retrieved context and generated response. `MemTrace` differs from these methods in its prediction target: rather than reporting component-level quality, it identifies a concrete faulty operation and explains why that operation causes the observed failure. For example, within EverMemOS, an incorrect sufficiency judgment, a failure in a subsequent retrieval step, and a reranking failure may all manifest as low context recall, but `MemTrace` distinguishes these underlying causes and points to the specific operation that requires correction. Moreover, unlike RAG-specific diagnostic methods, `MemTrace` is not restricted to RAG pipelines and can be applied to different memory-system architectures, including failures introduced during memory construction.

Operation-Level Memory Evaluation. Prior works such as HaluMem (Chen et al., 2025) evaluate extraction, update, and deletion operations against annotated golden memory points. This proactive evaluation is effective when intermediate supervision is available. `MemTrace` addresses a complementary setting: it starts from an observed end-task failure and traces backward through the actual execution graph using the question and reference answer, without requiring a golden memory state for every session or validating every memory operation in advance. It can also attribute failures in complex retrieval or response pipelines beyond predefined memory-maintenance operations.

C Non-Parametric Memory Systems

Consider a historical observation trajectory of length n, denoted as $\tau = \{m_i\}_{i=1}^{n}$, and an associated question-answer pair (q, a). Here, m_i denotes the i-th message in the trajectory, q is a question whose answer requires certain critical information from the historical observations, and a is the corresponding golden answer. We assume that the timestamp of the question satisfies $t_q > t_{m_n}$, meaning that the question is asked after all messages in the trajectory have been observed. Let $\mathcal{M}$ denote a non-parametric memory system, $\mathcal{U}_{\mathcal{M}}$ denote the memory update operation, and $\mathcal{R}_{\mathcal{M}}$ denote the memory read operation. During memory construction, the memory system incrementally updates its memory state in a message-by-message manner:

$$\mathcal{S}_j = \mathcal{U}_{\mathcal{M}}(\mathcal{S}_{j-1}, m_j), \quad 1 \le j \le n. \tag{4}$$

Given the question q at time t_q, the system reads relevant context $\mathcal{C}$ from the latest memory state $\mathcal{S}_n$ and generates an answer $\hat{a}$ based on the question-answering model $\mathcal{Q}$:

$$\begin{aligned} \mathcal{C} &= \mathcal{R}_{\mathcal{M}}(\mathcal{S}_n, q), \\ \hat{a} &= \mathcal{Q}(q, \mathcal{C}). \end{aligned} \tag{5}$$

Under this formulation, both long-context models and RAG systems can be viewed as instances of non-parametric memory systems. For long-context models, the update operation simply appends each new message to the context, and the read operation returns the entire memory state as the input context. For RAG systems, the update operation typically stores raw messages in a vector database, while the read operation performs top-K semantic retrieval. More advanced memory systems further introduce large language models into either the memory update operation, the memory read operation, or both. This increasing complexity introduces many sub-operations into memory updates and reads, making it difficult to find the source of errors.

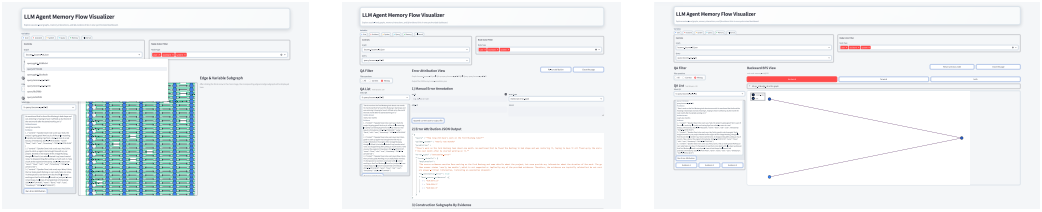


Figure 6: **The annotation interface with each thumbnail clickable and linking to its corresponding full-size visualization.** The left panel shows the entry point to the annotation interface, with the full-size visualization shown in Figure 13. The middle panel presents the annotation submission view, with the full-size visualization shown in Figure 14. The right panel provides an interface for exploring the execution graph, with the full-size visualization shown in Figure 15.

D Dataset Construction

D.1 Data Sources and Memory Systems

To obtain diverse interaction settings, we collect question-answer pairs from LoCoMo (Maharana et al., 2024), LongMemEval (Wu et al., 2025), and RealMem (Bian et al., 2026). These datasets differ substantially in their trajectory structure and evaluation protocols. In LoCoMo and LongMemEval, questions are posed after the interaction trajectory has been completed. LoCoMo provides multiple questions for each trajectory, whereas LongMemEval provides a single question per trajectory. By contrast, RealMem interleaves questions within the interaction, where each task query corresponds to a user message and the gold answer is the subsequent assistant response.

Accordingly, we use different evaluation protocols for these datasets when collecting execution graphs. For LoCoMo and LongMemEval, we adopt an offline protocol, where the full interaction trajectory is first processed by the memory system and all questions are evaluated afterward. For RealMem, we follow an online protocol. When a user message corresponds to a task query, the system answers the question using only the memory state available at that point, before the message is incorporated into memory.

We select four representative memory systems, namely long-context memory, RAG, Mem0, and EverMemOS. These systems cover a broad range of memory construction and retrieval mechanisms. The long-context baseline maintains the entire interaction history as a single memory unit and updates it with a rule-based procedure. RAG constructs an external memory store without any information loss. Mem0 supports not only memory addition but also memory update and deletion operations. EverMemOS uses a more complex retrieval pipeline involving LLM-based sufficiency judgment and query refinement.

Together, the heterogeneous source datasets and diverse memory systems provide a strong basis for collecting execution graphs that vary in both structural properties and failure modes.

D.2 Execution Graph Collection Details

To collect each memory system's execution graphs, we instrument each memory system's source code with `smartcomment` tracing statements. During instrumentation, we carefully choose the granularity of operations to make failure attribution meaningful and reduce annotation ambiguity. In some cases, we merge multiple low-level operations into a single traced operation when their failures are difficult to separate reliably. For example, in EverMemOS, retrieval involves an initial search, an LLM-based sufficiency judgment, optional query refinement, a second search, and result aggregation. If the system fails because key memory units are missing while the LLM judges the retrieved evidence as sufficient, it is ambiguous whether the error should be attributed to the initial retrieval or to the sufficiency judgment. Merging these tightly coupled steps into one traced operation avoids such borderline cases.

After instrumentation, we run all four memory systems on a sampled set of trajectories, including 4 from LoCoMo, 200 from LongMemEval, and 3 from RealMem. Each memory system processes each interaction trajectory message by message and produces one execution graph per trajectory.

The detailed system configurations are provided in Appendix D.3. In total, we collect 1,514 distinct errors across all systems after filtering out unanswerable questions[5]. Among them, RAG, long-context, Mem0, and EverMemOS account for 467, 279, 466, and 302 errors, respectively. The corresponding numbers of execution graphs are 138, 132, 87, and 80.

D.3 Experimental Setup for Memory Systems

We standardize the backbone models and evaluation settings across all memory systems to ensure fair comparison, particularly for subsequent failure mode analysis. Specifically, both the memory construction model and the downstream response generation model are set to GPT-4.1 mini (OpenAI, 2025). For dense retrieval, we use Qwen3-Embedding-4B (Zhang et al., 2025f) as the embedding model. For EverMemOS, the reranker is Qwen3-Reranker-4B (Zhang et al., 2025f). Both models are deployed using vLLM (Kwon et al., 2023). The number of retrieved memory units is fixed to 10 for all systems.

For evaluation, we adopt an LLM-as-a-Judge protocol (Gu et al., 2024) across all datasets, with dataset-specific prompts. For LoCoMo, the evaluation prompt is adapted from the Mem0 paper. For LongMemEval, we follow the official implementation. For RealMem, we adapt the official prompt by converting the original scoring scheme from a 0-3 scale to a binary format. The judge model is Claude Opus 4.5 (Anthropic, 2025), which helps mitigate evaluation bias due to architectural differences from GPT-based models (Zheng et al., 2023; Liu et al., 2024b).

For the long-context baseline, the context window is set to 32k tokens. For the RAG baseline, we adopt an online chunking strategy to construct memory units. Concretely, we maintain a message buffer, and when adding a new message would cause the total token count to exceed 1200, the existing buffer (if non-empty) is flushed before inserting the new message. The flushed messages are concatenated to form a memory unit, which is then added in the memory store. For Mem0, we use its open-source implementation without graph-based extensions, and retain the default prompts and configurations. For EverMemOS, we adapt the official evaluation pipeline. For boundary detection, the token limit and message limit are set to 8192 and 50, respectively. During hybrid retrieval, both sparse and dense retrievers return top-50 memory units. After the initial hybrid retrieval and reranking, the top 10 memory units are used for sufficiency checking. Multiple refined queries may be generated for the second-stage retrieval. For reciprocal rank fusion (RRF) in EverMemOS, we set the fusion constant to 60.

D.4 Annotation Process

We recruit five annotators from the author team to label failure information. All annotators have substantial experience with LLM agents and are familiar with the memory construction and retrieval mechanisms of the four target memory systems. Before the annotation process, we define an error taxonomy (Appendix D.5) and provide detailed annotation guidelines spanning more than 10 pages[6]. Given that each execution graph may contain thousands of nodes, direct inspection is challenging. To facilitate efficient and reliable annotation, we develop an interactive annotation interface (see Appendix D.6).

The annotation process consists of three stages. In the first stage, for each memory system, we randomly shuffle the collected erroneous cases and present them to annotators in a consistent order. Annotators then review the cases sequentially and stop once 40 system-related errors have been identified[7]. Each memory system is assigned to three annotators, who work independently without communication during this stage. For each erroneous case, annotators are required to specify the error type, identify the earliest faulty operation, and provide a natural-language explanation of the failure. After the first stage, a small fraction of cases may still have fewer than three annotations. In the second stage, we identify these cases and notify each annotator of the remaining cases they need

[5] We currently only consider questions with available source evidence. Unanswerable questions, by definition, do not have such evidence.

[6] Annotators are allowed to use their knowledge of each memory system to simplify annotation. For example, RAG does not introduce memory-construction errors in our setup, and long-context memory does not introduce retrieval errors because all retained context is provided directly to the response generator.

[7] Errors arising from annotation mistakes or LLM-as-a-judge issues are excluded from this quota, as they are not attributable to the system itself. Nevertheless, such errors are still recorded without an upper bound.

to label, ensuring that every annotated case receives labels from three annotators. In the third stage, for cases whose final label cannot be determined by majority voting, the annotators discuss the case collectively. During this discussion, we require annotators to revisit the execution graph and failure explanation until a consensus label is reached. Ultimately, this process yields 160 system-related failure cases, along with 67 annotation errors and 3 LLM-as-a-Judge errors.

D.5 Error Taxonomy

According to the lifecycle of information flow in stateful agents with non-parametric long-term memory, we define seven error types as follows.

Annotation Error. The source-evidence set associated with a QA pair is insufficient to support the reference answer, or the reference answer itself is incorrect with respect to the evidence.

LLM-as-a-Judge Error. The system's answer is in fact acceptable, but the automatic judge incorrectly marks it as wrong.

Extraction Error. The critical information is never captured into any memory unit during memory construction, and thus never enters the memory store.

Update Error. A memory unit initially contains the critical information, but a subsequent update operation modifies the unit in a way that removes or degrades that information.

Deletion Error. A memory unit containing the critical information exists but is explicitly removed (e.g., due to memory management), resulting in irreversible information loss.

Retrieval Error. At retrieval time, the memory store does contain the required information, but the retrieval pipeline fails to include it in the final retrieved context.

Response Error. The retrieved context contains all necessary evidence, yet the final LLM still produces an incorrect answer.

D.6 Annotation Interface Design

Figure 6 illustrates our annotation interface. Due to the large scale of execution graphs (averaging 2,262.65 nodes and 3,613.70 edges), it is impractical to display the entire graph at once. Therefore, the interface is designed to present only partial views tailored to the annotation workflow.

Since memory construction operations occur between pairs of messages, the entry panel focuses on explicit interactions between the user and the AI assistant. The lower-left section displays the current question-answer pair, including the question, golden answer, model response, and associated source evidence. Several control buttons are provided. One triggers an auxiliary error attribution algorithm, while others highlight the positions of corresponding source evidence within the message stream. The control panel in the upper-left corner shows the graph file associated with the current question-answer pair, along with a shuffled list of erroneous cases. Annotators are required to follow this predefined order during annotation.

Upon clicking the "Run Error Attribution" button, the system executes a preliminary automatic error attribution algorithm and transitions to the middle panel of Figure 6, which serves as the annotation submission interface. This panel presents the predicted error types and the locations of suspected faulty operations. These results are intended as guidance rather than definitive labels. In this interface, annotators can inspect multiple subgraphs to facilitate error localization. Specifically, they can examine (i) memory construction subgraphs induced by each source evidence[8], (ii) retrieval subgraphs, and (iii) response generation subgraphs. If further inspection is needed, annotators can click on any variable within these subgraphs to navigate to the execution graph exploration interface

[8] If a source evidence leads to the creation of a new memory unit, subsequent updates or deletions of that unit caused by other source evidence are not shown in the induced subgraph. Such downstream changes can be inspected in the execution graph exploration interface.

(the right panel in Figure 6). This interface supports both backward tracing (identifying the variables that produce the current variable) and forward tracing (tracking its downstream effects), enabling flexible exploration of the execution graph.

Overall, our annotation interface significantly improves annotation efficiency by structuring complex execution graphs into manageable, task-oriented views.

E TRACING TOOLKIT

`smartcomment` is a lightweight tracing package for recording developer-specified operations and the variables flowing through them. It is designed to collect execution graphs from existing Python systems without requiring developers to rewrite the original implementation around a new runtime abstraction.

E.1 COMPARISON WITH PRIOR TRACING FRAMEWORKS

A wide range of open-source frameworks are developed for collecting execution traces. Broadly speaking, these frameworks follow two design philosophies. The first philosophy is *instrumentation-based tracing*, adopted by systems such as MLflow (Zaharia et al., 2018), VizTracer (Gao, 2020), PySnooper (Rachum et al., 2019), and Langfuse (Langfuse Team, 2023). These frameworks ask developers to insert decorators, context managers, logging calls, or other hook-like statements at selected locations in the source code, in order to capture execution events such as function inputs, outputs, and auxiliary metadata. A major advantage of this design is that it is largely non-intrusive to the underlying application logic. Developers usually do not need to rewrite the data schema or reorganize the program around a new runtime abstraction. However, such frameworks are primarily event-centric. They can record spans, function calls, and metadata, but they generally do not make dependencies among intermediate variables first-class objects. As a result, they are less suitable for questions that require tracing how a particular memory unit is produced, how it is later modified, and which earlier variables causally contributed to a downstream failure.

The second philosophy is *abstraction-native tracing*, where computation is expressed through framework-specific data containers or operators that can be automatically intercepted and organized into graphs. Representative examples include TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), TextGrad (Yüksekgönül et al., 2025), Trace (Cheng et al., 2024), and DSPy (Khattab et al., 2023). These frameworks can often capture execution traces automatically and, in some cases, construct computation or dataflow graphs that encode dependencies among intermediate variables and operations. Their strength, however, comes from a strong assumption that the program must be written in terms of the framework's own abstractions. This is effective when the target domain already admits a stable computational substrate, such as tensors, modules, or other framework-defined objects. It is much less suitable for memory systems, whose schemas are heterogeneous, whose operations are highly flexible, and whose key entities (e.g., memory units, retrieval results, summaries, updates, and prompts) do not naturally fit into a single predefined abstraction.

These limitations motivate us to develop `smartcomment`. Conceptually, `smartcomment` combines the flexibility of instrumentation-based tracing with the provenance benefits of graph-based tracing. It uses explicit instrumentation, but instead of recording only events or call trees, it allows developers to trace arbitrary Python variables through user-defined serializable representations and to explicitly record dependencies among variables and operations. Both variables and operations can be annotated with comments and semantic metadata, which makes the resulting execution graph easier to inspect and interpret. In addition, `smartcomment` supports in-place updates through versioned variable nodes, allowing us to recover the evolution trajectory of a memory unit over time rather than only its final state.

This design is particularly useful for stateful agents with non-parametric memory, where one often needs to inspect not only what operation occurs, but also how a memory artifact is created, updated, retrieved, and eventually used in producing an answer. More broadly, `smartcomment` is not specific to memory systems. It can also be applied to other programs with rich evolving state, such as dynamic task planning or business data workflows. We view it as a general-purpose toolkit for collecting execution graphs that can support future research on automatic failure attribution and program understanding.

At the same time, `smartcomment` inherits the main trade-off of explicit instrumentation. Because the graph is constructed through developer-authored tracing statements, its quality depends on instrumentation coverage and granularity. Unlike framework-native tracers, it does not automatically capture all computations by default, and developers must define suitable representations and identities for the variables they wish to track. In this sense, `smartcomment` prioritizes flexibility and semantic expressiveness over full automation. This design choice is motivated by a broader shift in modern AI-assisted development environments, where the cost of writing and modifying code is significantly reduced. As a result, the traditional trade-off between ease-of-use and flexibility is shifting, increasingly favoring expressivity and composability over rigid, abstraction-heavy designs.

E.2 Design and Features of `smartcomment`

`smartcomment` has the following key features.

Hierarchical Data Model. `smartcomment` organizes traces using a hierarchical data model. At the top level, an execution graph stores the traced state of a program run. A graph contains sessions, which can be used to represent different stages of the system lifecycle, such as memory construction. Within each session, operations represent developer-specified computational steps, and variables represent the intermediate artifacts consumed or produced by these operations. Dependencies among variables are represented by edges associated with the corresponding operation. This design allows `smartcomment` to capture not only which operations are executed, but also how information flows across variables over time.

Explicit Instrumentation. Execution graphs are built through lightweight instrumentation in `smartcomment`[9]. Developers only need to insert a small number of tracing statements at key locations to mark operation boundaries, inputs, outputs, and dependencies. This makes it possible to trace existing memory-system implementations without restructuring their code. To recognize the same variable as it is passed, copied, or updated across different parts of the program, `smartcomment` uses a global tracing context and user-definable identity functions. They help determine when two runtime objects should be treated as the same traced variable. This is especially important for memory systems, where the same memory unit may be passed through multiple components, updated in place, or re-created from serialized representations.

Rich Contextual Attributes. `smartcomment` supports providing contextual information for sessions, operations, variables, and edges. Developers can assign category labels, natural-language comments, and custom metadata to each traced object. For example, for operations involving LLM calls, the metadata can record the model name, hyperparameters for text generation, or error messages returned by the API. These contextual attributes provide semantic context beyond raw execution events, making the resulting graphs easier for both humans and agents to inspect, interpret, and debug.

Persistence and Visualization. `smartcomment` supports graph export and import, enabling execution graphs to be persisted across program runs and restored for later analysis. It also provides visualization utilities based on PyVis[10] and Graphviz[11], which allow developers and annotators to inspect execution graphs interactively or as static diagrams. These capabilities are useful for validating instrumentation quality, and understanding complex system behavior.

E.3 Instrumentation Example

Figure 12 shows an instrumentation example in Mem0[12]. We insert two `smartcomment` statements into the memory-deletion method of the class `Memory` to record the deletion of a memory

[9] In the engineering implementation, the internal graph stored by `smartcomment` is not strictly a bipartite graph in the sense of Section 2. Variables are connected directly by edges, and each edge stores the identifier of the operation that induces the dependency.

[10] https://github.com/westhealth/pyvis

[11] https://github.com/xflr6/graphviz

[12] The full instrumentation details for all four memory systems are available in the released source code.

unit. Since deletion removes the original memory unit from the memory store, we introduce a constant deletion marker and register it as a traced variable using `comment_variable`. We then use `comment_link` to connect the deleted memory unit to this marker, explicitly representing the deletion effect in the execution trace.

The `comment_link` call is executed within the current operation context, so the resulting dependency edge is automatically associated with the corresponding operation identifier. In this example, the `source` argument of `comment_link` is specified as a Python tuple. The first element is a snapshot of the deleted memory unit represented as a Python dictionary. The second element provides tracing configurations and contextual attributes of this memory unit. The identity strategy used here is registered in `smartcomment` as `mem0-dict`.

F Multi-Error Attribution

Background. The problem formulation in Section 2 focuses on identifying a single earliest decisive faulty operation. In more complex systems, however, a failure may be jointly caused by multiple independent errors, such that correcting any one of them alone is insufficient. For example, a multi-user memory system may maintain separate memory stores for different users while answering a question requires combining information across several users' histories. Independent errors in these stores can then jointly cause the final failure.

Decisive Error Set. The execution-graph formulation can represent these cases through a **Decisive Error Set**. Let $O_c \subseteq \mathcal{O}$ be a candidate set of faulty operations. We say that O_c is valid if every operation in O_c is faulty, all operations in its strictly upstream ancestor set $\mathrm{Anc}_{\mathcal{G}}(O_c)$ are functionally correct, and O_c is causally sufficient. For the last condition, we construct $\mathcal{G}^{(O_c,*)}$ by replacing the faulty output variables of all operations in O_c with their correct counterparts and assuming ideal execution for all operations in $\mathrm{Desc}_{\mathcal{G}}(O_c)$. The candidate set is causally sufficient if this intervention rescues the execution, i.e., $Z(\mathcal{G}^{(O_c,*)}) = 0$. Let $\mathcal{F}(\mathcal{G})$ contain all candidate sets satisfying these conditions. The decisive error set $\mathcal{O}^*$ is then defined by imposing a minimality constraint over this feasible space: removing any operation from $\mathcal{O}^*$ breaks causal sufficiency. This is expressed mathematically as:

$$\mathcal{O}^* \in \big\{ O_c \in \mathcal{F}(\mathcal{G}) \mid \nexists O' \in \mathcal{F}(\mathcal{G}) \text{ s.t. } O' \subset O_c \big\}. \tag{6}$$

This formulation reduces multi-error attribution to identifying a minimal topological frontier of faulty operations that cause the system failure.

Extending `MemTrace`. Although current implementation of `MemTrace` targets singleton decisive error sets, the same framework can be extended to non-singleton cases. Once the agent identifies a decisive faulty operation, it can record it and prune operations strictly downstream of it, since these operations cannot belong to the same earliest decisive error set. The agent can then continue from the remaining to-explore list to search for additional independent faulty operations in the pruned execution graph. We leave this to future work.

G Additional Experimental Details for Failure Attribution

For both `MemTrace-OBS` and `MemTrace`, we set the working-context safety threshold to $T = 272{,}000$ tokens and the maximum reasoning budget to 200 iterations. We use Qwen3-Embedding-4B (Zhang et al., 2025f) as the embedding model used to initialize starting points. For `MemTrace`, the maximum to-explore list size is $N = 16$. For All-at-Once, we convert the execution graph into a weakly structured operation log. Because these logs are extremely long, we retain the latest 600,000 tokens and truncate earlier operations.

We also explore a long-context baseline inspired by MemAgent (Yu et al., 2025). It divides the operation log into chunks and processes them sequentially. After each chunk, the agent updates its working memory. After processing the complete log, it predicts the earliest decisive faulty operation using the final working memory. However, this approach must scan the entire trajectory, including

many failure-irrelevant operations. It is therefore prohibitively slow and performs poorly, so we exclude it from the comparison.

H Report Generation Details

We use GPT-5.4 to synthesize the error attribution results into a coherent error analysis report. The attribution results are obtained using different diagnostic settings for the two systems. For Mem0, we use the standard `MemTrace` setting. For EverMemOS, we leverage source evidence and prior knowledge. Specifically, we feed the model mini-batches of attribution outputs (with a batch size of four) and prompt it to iteratively update and refine the current report. When necessary, the model is also encouraged to identify finer-grained subtypes within each major error category. To improve the clarity and writing quality of the generated analysis, we include an exemplar error analysis section from the MMMU paper (Yue et al., 2024) in the prompt as an in-context example.

I Prompt Optimization

I.1 Prior Prompt Optimization Methods

Multi-session memory systems create very long chains between a prompt and the eventual failure it causes. For example, a fact extracted incorrectly in an early session may not lead to a visible mistake until hundreds of turns later. Before the error finally appears, the incorrect information may already have passed through many operations including memory update and memory retrieval. This property makes existing prompt optimization methods difficult to apply effectively.

Reflection-based optimizers. Methods such as ACE (Zhang et al., 2025c), and GEPA (Agrawal et al., 2025) feed the entire execution trajectory to an optimizer model. The optimizer is asked to reflect on the current performance and rewrite the prompts. In multi-session memory systems, the resulting trajectory exceeds the optimizer's context window. Even when the trajectory fits, optimizer attention degrades on long inputs, so reasoning over the relevant operations remains unreliable (Liu et al., 2024a; Modarressi et al., 2025).

Stage	Tokens	Time
`MemTrace`	493.21	1.33
Feedback Generation	17.19	0.23
Feedback Aggregation	16.25	1.23
Prompt Update	12.26	1.13

Table 4: Average cost of the closed-loop prompt optimization pipeline. "Tokens" denotes the average token cost, in thousands, including both input and output tokens. "Time" denotes the average end-to-end runtime in minutes. For `MemTrace` and feedback generation, averages are computed per failed case. For feedback aggregation and prompt update, averages are computed per target prompt variable.

Candidate-and-replay search. Methods such as OPRO (Yang et al., 2024a) and MIPRO (Opsahl-Ong et al., 2024) sample N prompt configuration candidates and score each by replaying the pipeline on a mini-batch of training set. However, re-running a memory system requires feeding the entire long interaction trajectory in an online manner, making the computational cost increasingly prohibitive as the number of prompt candidates grows.

Textual back-propagation. Methods such as TextGrad (Yüksekgönül et al., 2025) avoid replaying the trajectory by propagating natural-language feedback backward through the computation graph. However, because the computation graph is extremely long and many operations are unrelated to the failure, it is highly susceptible to signal blockage, downstream over-correction, and upstream pollution (Huang et al., 2026).

Common root cause. The three failure modes share a single root cause: each family attempts to reason about, propagate signals through, or replay the *entire* causal chain end-to-end. Our approach instead first localizes the faulty operation on the execution graph, reducing prompt optimization to a small, well-scoped sub-problem on which any of the three families above can be applied without modification.

Dataset	Sparse	Dense	Hybrid
LoCoMo	78.48	79.75	**89.87**
LongMemEval	70.27	79.73	**81.76**
RealMem	34.85	31.82	**39.39**
Overall	61.20	63.77	**70.34**

(a) The performance of different retrieval methods across datasets.

Retrieval Query	Recall@8
Query	59.50
Query + System Prediction	62.20
Query + Golden Answer	**70.34**

(b) Source-evidence retrieval performance under different query-construction strategies.

Table 5: **Retrieval performance analysis.** All values are reported as percentages.

I.2 Experimental Details

Setup. For Mem0, we optimize three prompts: the fact-extraction prompt used during memory construction, the memory-update decision prompt, and the question-answering prompt used at inference time. We assume a developer-in-the-loop setting in which the developer understands the target memory system and has access to data for iterative evaluation and improvement. The configuration for running Mem0 and collecting execution graphs follows Appendix D.3. Mem0 is initialized with its default prompt configuration. Both `MemTrace` and the prompt optimizer use GPT-5.4 as the backbone.

We randomly sample three LoCoMo user trajectories for optimization and reserve the remaining seven for testing. Optimization proceeds for three rounds. At round j, we run the current memory system on the j-th trajectory and evaluate it using the corresponding question-answer pairs.

`MemTrace`-Guided Optimization. Failed cases are passed to `MemTrace` for operation-level credit assignment. We initialize its to-explore list with the source evidence associated with each case and provide a high-level description of the Mem0 pipeline as prior knowledge. After `MemTrace` localizes the faulty operation, optimization is restricted to the prompts participating in that operation.

Because each localized operation subgraph is small, we use a lightweight prompt optimizer. It first generates feedback from the failure information and operation subgraph, aggregates the feedback across cases, and then rewrites the target prompt. Following TextGrad, the optimizer retains one previous version of each prompt variable. After each round, we update the corresponding Mem0 prompts before proceeding to the next trajectory.

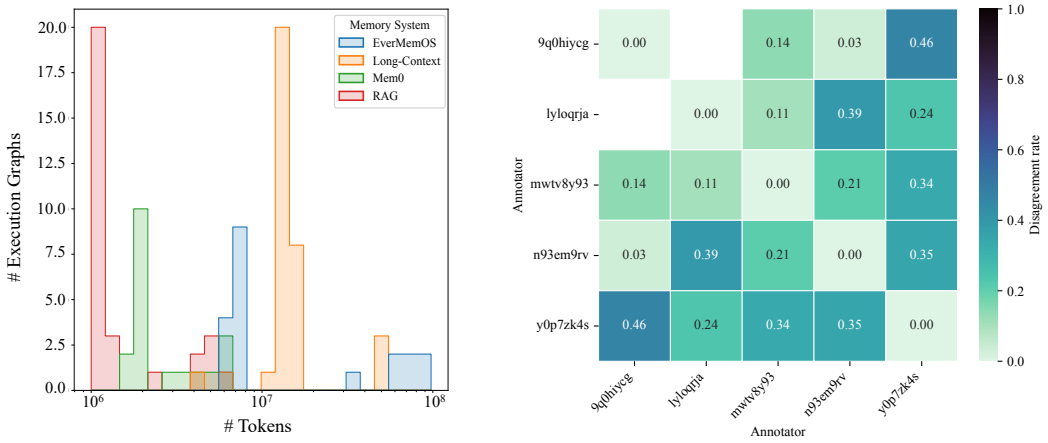
No-Attribution Baseline. To isolate the contribution of operation-level credit assignment, we construct a no-attribution baseline using the same optimization data, failure cases, backbone, optimizer, and three-round budget. This baseline bypasses `MemTrace` localization and provides the optimizer only with the final response-stage subgraph. Since no faulty operation is identified, the optimizer is asked to jointly update all three prompts. Therefore, the only difference between the two settings is whether localized credit assignment is available.

Optimization Cost. Table 4 reports the average cost of our closed-loop optimization pipeline. Overall, the cost is dominated by `MemTrace`. Nevertheless, its average wall-clock runtime is only 1.33 minutes per failed case, making it practical for an offline prompt optimization loop. The subsequent optimizer stages are lightweight because they operate on localized operation subgraphs and target only the prompts involved in the faulty operation.

J Additional Analysis

J.1 Retrieval Query Construction

Golden answers provide strong cues for retrieving source evidence, but they are generally unavailable when diagnosing failures in production. We therefore evaluate two deployment-compatible alternatives: using the user query alone and concatenating the query with the system prediction. The latter is always available in the execution trace, even when the prediction is incorrect. We keep the RRF-based hybrid retriever fixed across all settings and report source-evidence Recall@8.



(a) Token Distribution

(b) Annotator Disagreement

Figure 7: **Additional dataset analysis.** (a) Token distribution of execution graph logs for each memory system. (b) Pairwise disagreement rates among annotators. Darker colors indicate higher disagreement. The disagreement between annotators 9q0hiycg and lyloqrja cannot be computed because their annotated cases do not overlap.

As shown in Table 5b, the golden answer achieves the highest recall, as expected. Nevertheless, incorporating the system prediction improves Recall@8 from 59.50% to 62.20% over using the query alone. Although the prediction is incorrect, it may still contain relevant entities, or partial facts derived from the memory system, providing useful cues for locating the corresponding source messages. This effect varies across memory systems. Adding the system prediction improves retrieval on Mem0 and EverMemOS, but degrades it on RAG and Long-Context. For example, Recall@8 on Long-Context decreases from 70.92% to 64.42%. This suggests that, for advanced memory systems, model predictions provide a practical alternative when golden answers are unavailable. In production, we can further use user feedbacks to identify relevant source evidence, which we leave as an interesting direction for future work.

J.2 Additional Dataset Analysis

Memory system execution logs exceed the long-context window of current popular LLMs. As presented in Figure 7a, each system produces traces exceeding one million tokens, with more advanced systems approaching 10^7 tokens. Moreover, long-context memory further increases log size because each update operation records both the pre- and post-update context windows. As a result, the scale of these traces makes it difficult for practitioners to process directly with long-context LLMs for end-to-end inspection (see Table 1 and Appendix G).

Data annotation is intrinsically challenging. We compute pairwise annotator disagreement based on the first-round annotations. As shown in Figure 7b, annotators exhibit non-trivial disagreement, with pairwise disagreement rates ranging from 3% to 46%. This suggests that annotating failure attribution cases is intrinsically challenging, especially for Mem0 and EverMemOS cases.

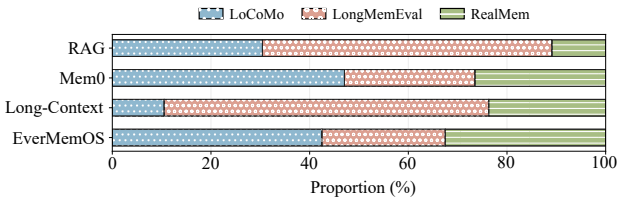


Figure 8: The dataset source distribution of system-related errors for each memory system.

System-related errors exhibit different dataset-source distributions across memory systems. As shown in Figure 8, the errors exhibit two different distribution

patterns across memory systems. For Mem0 and EverMemOS, LoCoMo contributes the largest share of system-related errors, whereas for RAG and long-context memory, most errors come from LongMemEval. One possible reason is that LongMemEval contains long user trajectories for each question-answering case, introducing more distracting context and making direct retrieval or long-context reasoning more difficult.

Systematic Error Analysis of Mem0

This analysis examines failure modes in Mem0 across memory construction, maintenance, retrieval, and downstream question answering, with the goal of understanding why conversationally available information does not reliably reappear as correct personalized recall. The failures show that robust performance depends not only on storing plausible memories, but on preserving the exact parts of an utterance that later become answer-bearing keys: temporal anchors, identifying nouns, updated numeric values, quoted formulations, evaluative wording, conversational next steps, completion states, speaker-specific commitments, and source details such as where an item was obtained. Across the cases, four recurring weaknesses stand out. Memory updates can corrupt previously correct facts by rewriting temporal grounding, collapsing repeated but distinct events into a single record, or stripping away relational and provenance details needed for later comparisons or literal recall. Fact extraction can prevent critical information from ever entering memory, especially when the decisive content is a relative date, a timing clause, a negative preference, a short evaluative phrase, a specific emotion, a structured plan, or an assistant-provided list that later becomes the target of recall. Retrieval based on the raw user question often favors semantically adjacent or person-related memories while omitting the exact memory whose wording, entity label, recency, completeness, or set membership resolves the query. Finally, even when relevant evidence is present, the answering model does not consistently use it to perform literal recall, enumerate all qualifying items, or preserve necessary temporal and state distinctions, and instead drifts toward plausible but unsupported continuations or incomplete counts.

A first category of failures arises from memory maintenance errors, particularly update operations that preserve surface topicality while silently overwriting essential event structure or fact specificity. In one pattern, **a memory can remain textually similar yet have its temporal grounding reassigned to a later timestamp**, as when an artwork memory about something created "last week" was rebound from the correct August context to October. In another pattern, **repeated events with similar schemas are wrongly merged instead of stored as separate episodes**. Dave's car-show history illustrates this clearly: an earlier memory that he went to a car show "last weekend" was updated into "last Friday" after a later message about a different car-show visit, effectively replacing one attendance event with another. A related subtype is specificity-degrading update. **Here the system starts from a locally correct memory and then rewrites it into a broader, less answerable version**, as when "User has opened their own car maintenance shop" became merely "Opened a shop which was a dream and involved a lot of hard work". The newer travel case showed that this degradation can remove relational detail needed for comparative reasoning, and the tennis-racket case shows an adjacent provenance-loss variant in which an update deletes the exact source field that later answers a "where from" question. "Has a new tennis racket from a sports store downtow" was rewritten into "Currently focusing on improving tennis game with a new racket", preserving topic but erasing the purchase location. This is not just stylistic drift. Once update decisions treat memories as mutable summaries rather than protected records with answer-bearing fields, the store loses timestamp integrity, event cardinality, lexical specificity, relational comparability, and source provenance, making later recall systematically unreliable.

Table 6: **Full systematic error analysis report automatically generated for Mem0.** Red bold text highlights major recurring failure mechanisms. Since `MemTrace` is not perfectly accurate, some identified errors may contain minor inaccuracies. However, from an overall perspective, `MemTrace` still provides meaningful insights.

Systematic Error Analysis of Mem0

A second major category consists of extraction failures, where decisive information never becomes retrievable because it is not converted into memory units at all, or is converted only in a weakened form. **One subtype is policy-induced omission: the extractor is explicitly instructed to store only user-message facts, so assistant-provided content such as a previously given example answer, a detailed plan, an itinerary revision, a recipe ingredient list, a delivered module breakdown, a venue recommendation list, or the declared next component after a milestone is dropped even though later questions ask the system to remember exactly what the assistant had said or already provided.** The Portland music-venue failure makes this especially clear: the assistant's full ordered list ending with "Revolution Hall" was fully present in the conversation but extraction returned an empty list, so later recall became impossible. A particularly important form of this mismatch is assistant-originated state/progress omission, where the omitted content is not merely descriptive but records that something has already been pushed, locked in, delivered, or advanced to the next stage; once that state transition is absent from memory, later responses regress to outdated planning talk or redundant re-explanation. **A second subtype is subtle user-fact omission, where the extractor returns no facts even though the user states a durable preference, constraint, or evaluative takeaway that is crucial for future responses. A third subtype is temporal-clause omission: when a message contains both a descriptive fact and a date-bearing or relative-time phrase, extraction tends to retain the high-level description while dropping the temporal detail that later becomes the direct target of a "when" question. A fourth subtype is evaluative-detail omission in socially directed messages, where praise, characterization, or causal attributions are compressed into generic encouragement or discarded altogether. A fifth and increasingly important subtype is structured-payload omission. In these cases the source message contains a long but highly answerable payload—such as a workout protocol, a route update, direct manifestations of a narrative pressure point, a paper breakdown, an ingredient list with exact quantities, or an ordered venue list—yet extraction returns an empty list because the content is not framed as a simple user profile fact.** A broader extraction pathology is answer-bearing lexical compression. **Here the extractor preserves topical gist but drops the exact word, phrase, or structured item that the later question asks for.** Dave's statement that fixing cars "makes me proud" was reduced to feeling "great", and his description of working on cars as "like therapy" was rewritten into a more generic engine-focused formulation. These are not harmless paraphrases. Because Mem0 later searches with raw user questions, losing exact emotion terms, business labels, milestone transitions, module-delivery states, numeric quantities, schedule revisions, ordered list items, or procedural details weakens both retrieval match quality and answer precision.

Retrieval errors remain a major source of failure and reveal several related weaknesses in using the raw question as a direct search query over fragmented memory units. **One recurring pattern is companion-memory omission, where the system retrieves a memory describing the focal event but not the adjacent fact needed to answer the question fully. Another is semantic genericity failure, in which broad memories about the same person, topic, or activity outrank the decisive, more specific memory.** Questions about counseling services, maintenance-program follow-up, networking plans, detailed workout instructions, KudiFlow route updates, and clothing pickup obligations all show that topical overlap alone is insufficient: generic memories about support, planning, flexibility, wardrobe organization, route changes, or KudiFlow tips can dominate while the single answer-bearing next step, named entity, actual plan payload, or locked-in revision is absent. **A third subtype is identifying-label omission under dense topical competition.** In the Ferrari case, the store already contained the decisive memory that Calvin had recently gotten a Ferrari and called it a "masterpiece on wheels", yet retrieval returned only vague vehicle-related, motivational, and person-specific memories. A fourth subtype is aggregate-set omission. **When the question asks for a list or inventory of activities or obligations, retrieval does not reliably surface the multiple distinct episodic memories needed to compose the set.** In the friend-activities case, the store contained separate memories for park walks, a countryside road trip, a shop-opening celebration, and card-playing nights, yet the retrieved context consisted of off-target items such as catch-up plans, generic positive interactions, and self-expression through cars. The clothing-store case shows the same failure in obligation form: the system retrieved the blazer dry-cleaning pickup but omitted both Zara boots memories, so a question requiring the total number of items to pick up or return was collapsed to a single item. **A fifth subtype is stale-memory preference despite available updates.** In the bird-species case, retrieval surfaced the older count of 27 and a related Northern Flicker memory, but omitted the later memory explicitly storing the updated total of 32. A sixth subtype, highlighted by the guide-editing failure, is status-contrast omission. Some questions do not merely ask for a topic but for the current state of multiple requested edits, such as whether one component has already been added while another has not. In those cases, **retrieval surfaces positive-progress memories around the project but misses the decisive contrasting evidence that distinguishes completed from not-yet-completed items**. Across these cases, direct vector search appears biased toward person identity, broad topical similarity, and diffuse semantic neighborhoods, while underweighting answer-bearing nouns, updated totals, quoted aliases, enumerative coverage, revision state, milestone transitions, event-to-consequence links, and the need to retrieve complete sets of related obligations or events rather than just the most individually similar item.

Table 7: The continued part of Table 6.

Systematic Error Analysis of Mem0

Question-answering errors show that evidence inclusion alone is not sufficient unless the answering model reliably identifies, combines, and computes over the memories that directly resolve the question. **One subtype is evidential selection failure, where the model prefers a more elaborate but less relevant narrative over a concise decisive memory, producing answers that are thematically plausible yet unsupported by the best evidence in context. Another subtype is task-mapping failure: even when the retrieved context contains the right project state and next-step cues, the model may answer a neighboring but different question by drifting toward another salient thread.** The script-development case is illustrative: the context already contained "optimizing engagement through pacing" and "ensure efficient data throughput in the story", yet the answer shifted to an unrelated Act III resolution task rather than using the retrieved evidence to name pacing as the next priority. **A third subtype is temporal grounding failure: when context contains relative-time expressions paired with message timestamps, the model may answer with the message date itself or anchor to a different but topically related memory rather than converting the relative expression into the correct calendar interval.** The art-events case highlights **a further subtype that is better characterized as aggregation and counting failure**. There, the final context already contained multiple qualifying art-event memories within the target window, but the model enumerated only one event and concluded the answer was 1 instead of 4. This shows that even when retrieval succeeds, the QA stage may fail to scan the full context, apply the query's inclusion criteria consistently, or aggregate all matching memories into a correct total. The failures also continue to show relative-time arithmetic and update-tracking weakness even when the needed evidence is partly present, with the model tending to copy the most salient duration string or stale numeric value rather than perform the small subtraction, filtering, or state resolution required by the question. More broadly, when the exact memory is absent or degraded, the QA model falls back to semantic reconstruction. In the maintenance-program case it improvised a plausible checklist rather than recalling the established next step; in the Ferrari case it inferred only that Calvin got some unspecified vehicle; in the emotion case it reproduced the extractor's weakened wording and answered "great" instead of "proud"; and in the workout-plan case it generated a generic seven-day routine from high-level fitness constraints rather than reproducing the stored three-day protocol. The guide-editing failure reveals a sharper completion bias: when retrieved context contains evidence that progress has been made on a project but omits the explicit memory that some requested detail is still missing, the model tends to over-affirm full completion rather than preserve partial status. These errors are often downstream manifestations of earlier pipeline failures, but they also reveal an independent weakness: the model does not consistently distinguish literal recall from plausible thematic synthesis, nor does it reliably perform complete set aggregation or abstain when the retrieved evidence lacks the exact answer.

Overall, the failures point to a pipeline whose reliability depends on aligning memory construction policy with the kinds of conversational recall it is expected to support, preserving episode identity and timestamp integrity during updates, protecting answer-bearing specificity during paraphrase, extracting structured and assistant-originated content rather than only high-level user summaries, retrieving the minimal set of decisive memories rather than semantically adjacent background facts, and constraining generation to explicit evidence and simple verifiable inference. The most important implication is that errors are often compositional: a restrictive extraction policy can make later recall impossible, a paraphrastic extraction can preserve topic while erasing the crucial word, a null extraction on a long structured or enumerated message can prevent any of the answer-bearing payload from entering memory, an update can silently collapse two distinct experiences into one or strip away the relational, lexical, or provenance detail that supports comparison and literal recall, a retrieval layer optimized for topical similarity can still miss the memory that explicitly names the answer, contains the latest value, or completes the relevant set, and a QA model can still ignore or undercount the correct evidence even when it is present. Effective personalized QA therefore requires better handling of assistant-originated conversational content when relevant, stronger preservation of temporal expressions, numeric updates, negative knowledge, evaluative wording, exact emotion terms, ingredient quantities, itinerary revisions, delivery states, venue lists, purchase sources, and next-step plans, update rules that protect event boundaries as well as lexical, relational, and provenance specificity, retrieval mechanisms that reward answer-bearing recency, completeness, set coverage, and state contrast over generic semantic proximity, and more conservative answer synthesis that explicitly acknowledges when the stored evidence does not support exact recall or when light filtering, counting, or aggregation is required.

Table 8: The continued part of Table 6.

Systematic Error Analysis of EverMemOS

This analysis examines failure modes in EverMemOS across memory construction, retrieval control, sufficiency assessment, and downstream answer generation, with the aim of identifying the system behaviors that most often break end-to-end factual recall. The dominant errors do not arise simply because information was never stored. Instead, many failures reflect losses of precision at stage boundaries: the system may preserve the decisive memory in storage, yet retrieve the wrong stage of the user's plan, judge incomplete evidence as sufficient, or pass a misleading context to the answer model. Across the observed cases, the most consequential weakness is therefore not raw memory coverage but state fidelity. EverMemOS frequently retains enough topical material to sound plausible, while missing the exact fact, relation, temporal anchor, commitment status, or latest event update needed to answer correctly.

A major class of errors comes from response-stage evidence misuse, where the retrieved context already contains the key information but the question-answering model still produces the wrong answer. These failures appear in several recurring forms. **One is competitive salience failure, where the model latches onto a nearby but non-answer-bearing topic and substitutes it for the requested target. Another is schema mismatch, where the model answers under the wrong counting or comparison frame, such as converting a question about quantities of items into a count of actions. A third is attribute misbinding, where the right cluster of candidate facts is present but a distinctive attribute, example, or description is attached to the wrong entity. A fourth is premature commitment on unresolved states: when memory preserves an open choice or a not-yet-completed action, the answer model collapses that uncertainty into a falsely finalized plan.** These patterns show that EverMemOS can often deliver enough evidence to support the answer in principle, yet the final generation stage remains brittle when success depends on preserving exact semantic roles and decision states rather than producing a topically plausible response.

A second major category involves retrieval steering and evidence-selection failures inside the adaptive retrieval pipeline. In some cases, relevant memories are stored correctly and even appear in the broader hybrid candidate pool, but the system's controller elevates the wrong evidence into the sufficiency-check set or final QA context. **One subtype is stale-state exclusion: the decisive memory is the latest update to a schedule, booking status, or plan state, but reranking favors older, topically similar memories, causing the system to answer from superseded evidence.** This is especially damaging for questions about what is next, what is already booked, or whether a plan has been confirmed, because older preparatory memories remain semantically close while differing critically in status. **A second subtype is off-target Round-2 query generation.** When Round-1 evidence is judged insufficient, EverMemOS can misdiagnose what is missing and generate follow-up queries that overfit to salient but irrelevant fragments of the current pool. In the clearest cases, the system imports the wrong retrieval schema entirely—for example, treating a non-temporal planning query as though it were a temporal-interval reconstruction problem—and then expands follow-up queries around downloading, scheduling, or setup details instead of the actual revision-decision episode. These failures reveal that retrieval quality is constrained not only by candidate recall, but by how the system interprets insufficiency and decides which latent subproblem to search for next.

Table 9: **Full systematic error analysis report automatically generated for EverMemOS.** Red bold text highlights major recurring failure mechanisms. Since `MemTrace` is not perfectly accurate, some identified errors may contain minor inaccuracies. However, from an overall perspective, `MemTrace` still provides meaningful insights.

Systematic Error Analysis of EverMemOS

A third recurring category is temporal and state-tracking fragility, which cuts across retrieval, sufficiency checking, and answer generation. Many questions depend not merely on topical relevance but on recovering the correct event state at the correct point in time: distinguishing current schedule from prior schedule, confirmed booking from intended booking, or today's planned workout from a remembered but misaligned future reference. EverMemOS struggles with several variants of this problem. **It can confuse mention time with event time, fail to privilege the most recent update over earlier planning memories, or collapse distinctions between intention, recommendation, and completion.** In other cases, **the sufficiency check itself is too coarse: it marks evidence as sufficient because the topic domain is covered, while overlooking a missing temporal or status-bearing fact that is essential to the question.** This leads the system to terminate retrieval early and answer from an incomplete context, as when intent to book campsites is treated as equivalent to campsites already being secured. These failures suggest that EverMemOS represents topical continuity more robustly than event-state transitions. It often remembers what the conversation was about, but not whether a key action was proposed, scheduled, postponed, completed, or still pending.

Overall, the error patterns indicate that EverMemOS's central challenge is not simply retrieving memories, but preserving the correct event state through retrieval control and answer grounding. The system is most reliable when a question can be answered by quoting a single salient fact, and least reliable when success depends on selecting the latest state update, distinguishing intention from completion, keeping unresolved decisions open, or generating follow-up retrieval plans that target the actual missing information rather than a plausible but irrelevant subtheme. Improving end-to-end performance will therefore require tighter coupling between retrieval, sufficiency judgment, and answer generation. Retrieval should optimize for evidential completeness at the level of event states and state transitions, not just topical relevance; the sufficiency checker should be more explicitly constrained to detect missing status and recency distinctions; and the final answer stage should be forced to ground claims in the retrieved temporal and decision structure. Without such improvements, the system will continue to fail even when the decisive memory is already present somewhere in storage or the initial candidate pool.

Table 10: The continued part of Table 9.

Question

How many points do I need to earn to redeem a free skincare product at Sephora?

Source Evidence:

- User (2023-05-21 12:19:00): I'm looking for some advice on skincare products. I recently bought an eyeshadow palette at Sephora and earned 50 points, bringing my total to 200 points so far in their loyalty program. Can you recommend some popular skincare products that would complement my eyeshadow purchases?
- User (2023-05-29 08:31:00): Do you know if Sephora has any current promotions or discounts on the La Roche-Posay moisturizer or any other products I might want to purchase with it? By the way, I'm really close to redeeming a free skincare product from Sephora, I just need a total of 300 points and I'm all set!

Golden Answer

100

Model Prediction

Based on your memories, you need to earn 300 points to redeem a free skincare product at Sephora. You currently have 200 points, so you are close but need 100 more points to reach the redemption threshold.

Annotation

Error Type: LLM-as-a-Judge Error
Reason: The query was somewhat ambiguous because it could refer either to the total redemption threshold or to the additional points needed from the user's current balance. The evidence showed that the user currently had 200 points and needed 300 total points to be eligible, meaning 100 more points were required. The golden answer only gave 100, but the prediction included both the total threshold of 300 points and the remaining amount of 100 points. These figures were accurate and supported by the evidence. The judge likely erred by treating the extra correct context as wrong rather than recognizing that the prediction contained the golden answer.

Figure 9: **An LLM-as-a-Judge error case.** Due to ambiguity between the total required points and the remaining points needed, the prediction provides both correct values ("300 total" and "100 remaining"), but the LLM judge marks it incorrect.

Question

How many games has John mentioned winning?

Source Evidence:

- John (2023-07-16 16:21:00): Thank you! Scoring those points was an incredible experience. The atmosphere was electric, and my teammates and I were thrilled. We pulled off a tough win!
- John (2023-08-09 10:29:00): Hi Tim! Nice to hear from you. Glad you could reconnect. As for me, lots of stuff happened since we last talked. Last week I had a crazy game - crazy intense! We won it by a tight score. Scoring that last basket and hearing the crowd cheer was awesome
- John (2023-12-08 19:42:00): That frog looks yummy! I haven't had one in ages. Been having some wild games lately, we played a top team and it was tough, but we fought hard and got the win! It's awesome having my team to push us all.
- John (2023-12-11 20:28:00): I've had some thrilling games in my career. My favorite was when we were down 10 in the 4th and I hit the buzzer-beater shot to win. The atmosphere was incredible and it was such a thrilling experience. Those moments make me love basketball so much.
- John (2023-12-16 15:37:00): Hey Tim! Nice to talk again. The b-ball games have been crazy. We had a real battle against another team last week. It was close until the final buzzer but we got the win.

Golden Answer

6

Annotation

Error Type: Annotation Error
Reason: The listed source evidence does not support the golden answer of 6 wins. It contains five distinct evidence entries, each describing one John win, for a total of five supported wins. Even if all listed evidence were retrieved correctly, it would only justify an answer of 5. The sixth win required by the golden answer is not present in the annotated evidence. Therefore, this is an annotation error rather than a memory-system operation error.

Figure 10: **An Annotation case from LoCoMo.** The evidence contains only five supported wins, but the golden answer expects six.

Question

I mentioned an investment for a competition four weeks ago? What did I buy?

Source Evidence:

- User (2023-03-04 13:12:00): I actually got my own set of sculpting tools, including a modeling tool set, a wire cutter, and a sculpting mat today. I'm excited to experiment with these new tools and techniques. Can you suggest some specific ways I can incorporate eco-friendly materials and techniques into my sculpting process, considering the tools I already have?

Golden Answer

I got my own set of sculpting tools.

Annotation

Error Type: Annotation Error
Reason: The evidence supports that the user bought or got their own set of sculpting tools. However, it does not mention that this purchase was an investment for a competition. The query requires a link between the purchase and a competition-related investment, but that context is absent from the source evidence. Therefore, the annotation is flawed because the gold answer is only partially supported by the evidence.

Figure 11: **An Annotation case from LongMemEval.** The evidence confirms that the user obtains sculpting tools, but does not support the claim that the purchase is a competition-related investment. The question should instead remove the investment-related wording and ask a simpler supported query such as “What did I buy four weeks ago”.

mem0/memory/main.py

```python
def _delete_memory(self, memory_id):
    logger.info(f"Deleting memory with {memory_id=}")
    existing_memory = self.vector_store.get(vector_id=memory_id)
    prev_value = existing_memory.payload.get("data", "")
    self.vector_store.delete(vector_id=memory_id)

    delete_marker = comment_variable(
        "DELETED",
        to_runtime=True,
        comment="It marks the memory unit is deleted successfully.",
        class_name="delete_marker",
        category="marker",
    )
    comment_link(
        source=(
            {
                "id": memory_id,
                "memory": prev_value,
                "timestamp": existing_memory.payload.get("timestamp"),
            },
            {
                "id_strategy": "mem0-dict",
                "category": "memory_entry",
                "encoding_fn": partial(
                    json.dumps,
                    ensure_ascii=False,
                    indent=4,
                    sort_keys=True,
                ),
                "decoding_fn": json.loads,
                "comment": "A memory unit in the memory system.",
            },
        ),
        target=delete_marker,
        comment=(
            f"The memory unit '{memory_id}' is deleted successfully."
        ),
    )

    self.db.add_history(
        memory_id,
        prev_value,
        None,
        "DELETE",
        actor_id=existing_memory.payload.get("actor_id"),
        role=existing_memory.payload.get("role"),
        is_deleted=1,
    )
    return memory_id
```

Figure 12: **An instrumentation example.** We insert two `smartcomment` statements, highlighted in green, into the method `_delete_memory` of the class `Memory` in the Mem0 source code to record the deletion of a memory unit. The method is extracted for presentation, with surrounding code omitted and indentation adjusted for readability.

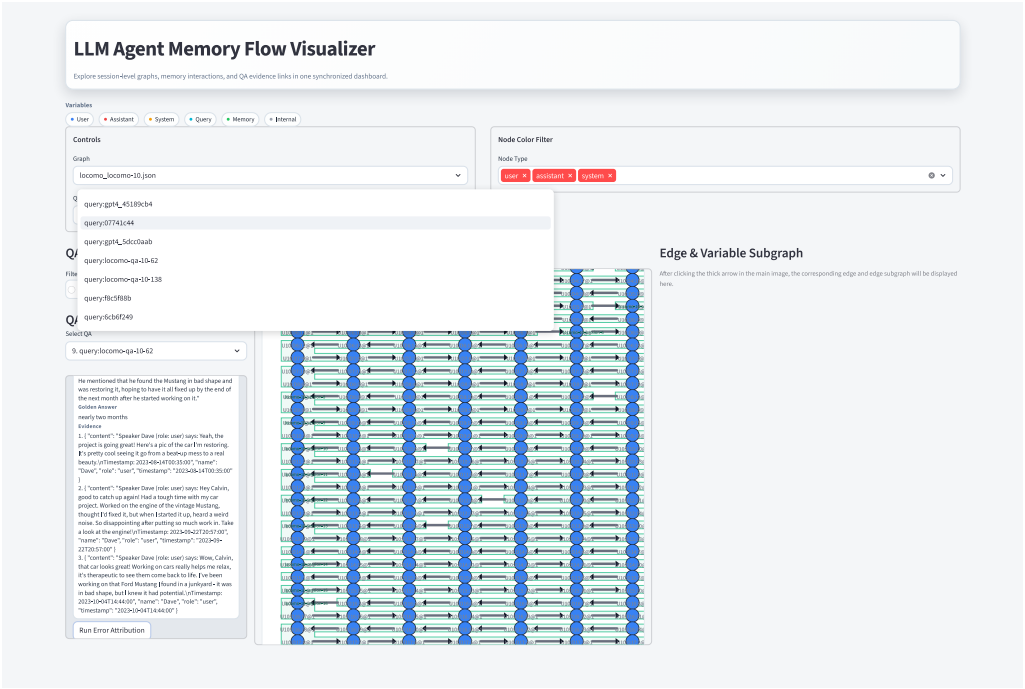


Figure 13: **Full-size visualization of the annotation interface entry point.** This figure corresponds to the left thumbnail in the overview shown in Figure 6.

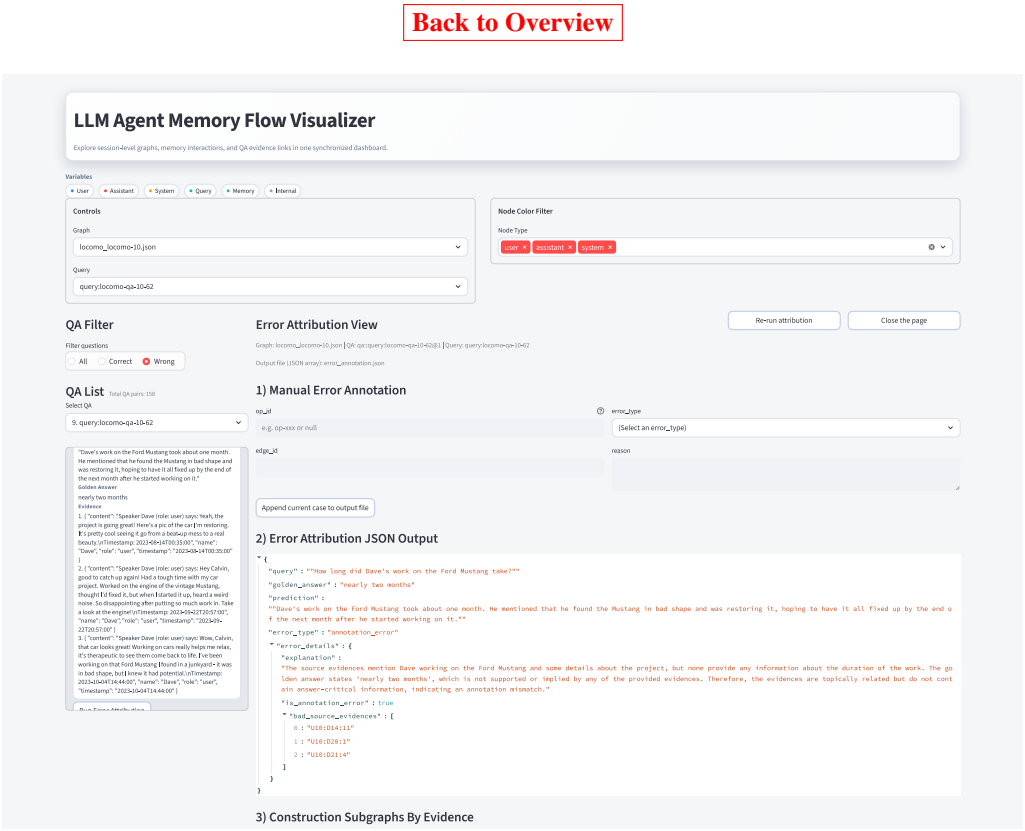


Figure 14: **Full-size visualization of the annotation submission view.** This figure corresponds to the middle thumbnail in the overview shown in Figure 6.

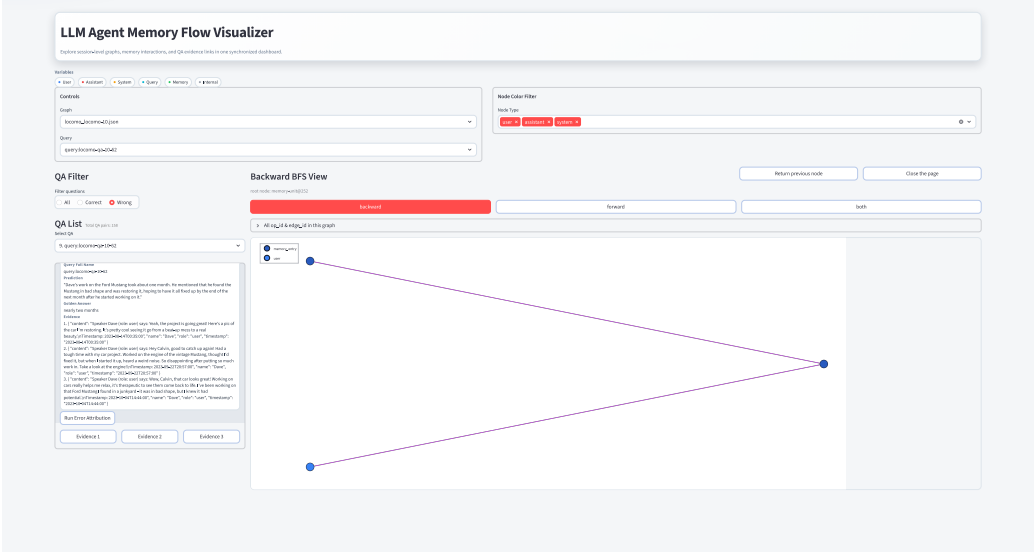


Figure 15: **Full-size visualization of the execution graph exploration interface.** This figure corresponds to the right thumbnail in the overview shown in Figure 6.