WHEN CORRECT MEMORY GOES WRONG: FUZZING PERSISTENT MEMORY USE IN LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Persistent memory helps LLM agents carry information across long interactions, but correct memory can still be used incorrectly when queries change or memory states evolve. Existing work mainly studies memory content errors or evaluates fixed test cases, leaving memory-use failures hard to discover systematically. We formulate this issue as a fuzzing problem and categorize such failures into query-related and memory-state failures. We then develop U-FUZZ, which starts from memory checkpoints as test seeds, mutates queries or memory states under explicit mutation obligations, validates each mutant, and uses observed memory behavior to guide iterative testing while keeping failure labels outside the search. We evaluate U-FUZZ across several memory systems against diverse fuzzing baselines, and further test an output-only setting with API-based LLMs where memory retrieval is hidden. Across these settings, U-FUZZ consistently uncovers more confirmed memory-use failures, showing that its search remains effective across different memory architectures and even when only final responses are observable[1].

1 INTRODUCTION

Persistent memory is widely used as the external storage of LLM agents to carry information across long and complex interactions, allowing agents to recall stored facts, past experiences, and user context when handling new requests (Maharana et al., 2024; Wu et al., 2025; Hu et al., 2026). Notably, **a correct memory can still be used incorrectly**: a small change to a request, such as paraphrasing, may cause the correct memory to be missed or displaced by another record, while a memory update may leave an outdated record ranked above the current one. Reliable agentic memory therefore depends on trustworthy stored content and robust use of that content.

However, existing work has mainly studied memory reliability of stored content. Security studies examine how poisoned or injected records enter memory and later affect agent behavior (Chen et al., 2024; Dong et al., 2026). Diagnostic frameworks further separate retrieval failures from failures in using retrieved memories (Yuan et al., 2026), and metamorphic testing has begun to test how RAG systems respond when their underlying corpora change (Kim et al., 2026a). These studies reveal important memory failures, but they provide limited ways to actively search for cases where an agent uses valid memory incorrectly. Such failures are especially practical because they can arise from ordinary user requests and routine memory updates, even when every stored item is free from poisoning or other failures (Xiong et al., 2026). As a result, inspections on memory items cannot detect the failure of memory use.

This work. We highlight that finding memory usage failures is essentially a fuzzing problem, as the same memory system can behave differently across many valid forms of a request and many memory states, while only a small part of this space is covered by test queries (Böhme et al., 2016; Xia et al., 2024; Kim et al., 2026b). It is therefore difficult to know in advance which query or memory change will trigger a failure. This motivates a fuzzing-based testing that dynamically generates new cases, observes the resulting memory behavior, and uses that feedback to decide what to test next.

To better support fuzzing design, we first categorize memory-use failures based on what changes during testing. The first group contains *query-triggered* failures, where the memory state is fixed but the request is adapted. For example, two paraphrases asking for the same fact may retrieve

[1]Code is available at `https://anonymous.4open.science/r/U-Fuzz-469C`.

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.

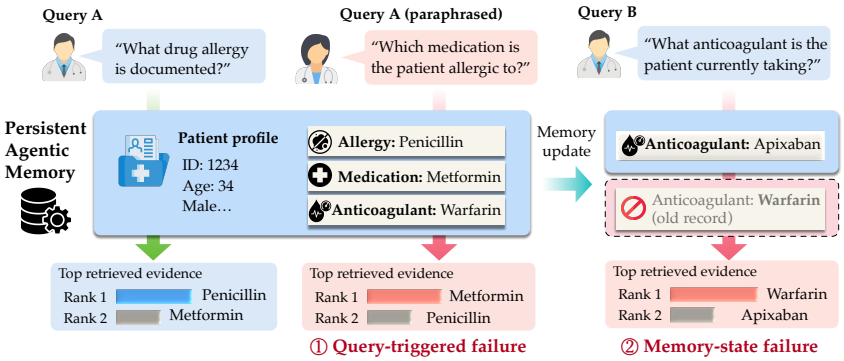


Figure 1: Illustrations of two memory-use failures: (1) paraphrasing a query changes the evidence retrieved from the same memory state; (2) after a memory update, stale evidence is ranked above the current value.

different memories, or a request for unsupported information may still retrieve an unrelated record. The second group contains *memory-state* failures, where the request is fixed but the stored memory changes. For example, an update may fail to replace an older value, a conflicting record may be handled incorrectly, or a deleted memory may still be retrieved. This categorization follows the two main factors that determine memory use: what the user asks and what is currently stored in memory. It also gives a simple way to test whether memory use remains correct when either factor changes.

To find these memory-use failures, we develop U-FUZZ, a fuzzing framework for persistent agentic memory. U-FUZZ starts from memory checkpoints built through normal interactions and constructs test seeds from their associated queries. It then mutates one input to memory use at a time, covering both query changes and controlled memory-state changes. Query mutations preserve the request, redirect it to another supported target, or make it unsupported; memory-state mutations update, delete, or modify unrelated stored information. Each mutant is validated before execution, while mutation obligations ensure that applicable operator-target combinations are exercised systematically. Observed execution behavior then guides which mutants are retained and explored in later fuzzing rounds, without using failure labels to direct the search.

We evaluate U-FUZZ on diverse memory systems (Chhikara et al., 2025; Xu et al., 2026; Rasmussen et al., 2025; Kang et al., 2025) using benchmarks (Maharana et al., 2024; Wu et al., 2025). Our experiments study four aspects. First, under the same execution budget, U-FUZZ consistently finds more unique memory-use failures than diverse baseline methods with different searching mechanisms. The results also show that broad memory coverage alone is not sufficient: distinct retrieval behavior within already reached memory remains important for failure discovery. Second, query and memory-state mutations exhibit different search dynamics and expose complementary failures; combining them helps U-FUZZ continue exploring unique memory-use failures in a long-term execution. Third, ablations show that the mutation operators contribute differently across memory backends, while retrieval feedback drives broader exploration or refines the search within explored memory. Finally, we evaluate an output-only setting in which retrieval is hidden, wherein U-FUZZ continues to find more unique and end-to-end failures than baselines, showing that direct retrieval access improves search guidance and diagnosis but is not required for memory-use testing.

Overall, we make three contributions:

- **A categorization of memory-use failures.** We distinguish query-related and memory-state failures, depending on whether incorrect memory use is triggered by a change in the request or in the stored memory.
- **A fuzzing framework for persistent memory.** We develop U-FUZZ, which systematically mutates queries and memory states, validates each mutant, and uses observed execution behavior to guide iterative testing without using failure labels during search.
- **Experimental evaluation.** We evaluate U-FUZZ across multiple memory systems and benchmarks, including an output-only setting with hidden retrieval, and show that it discovers memory-use failures missed by simpler testing strategies.

2 Related Work

Memory systems and evaluation. Persistent memory enables LLM agents to retain, retrieve, update, and organize information across interactions, with designs ranging from episodic memory to structured, dynamically linked, and hierarchical stores (Shinn et al., 2023; Chhikara et al., 2025; Xu et al., 2026; Kang et al., 2025). Long-horizon benchmarks evaluate these systems through factual recall, temporal and multi-session reasoning, knowledge updates, abstention, and selective forgetting (Maharana et al., 2024; Wu et al., 2025; Hu et al., 2026). Recent empirical work further examines how memory-management operations affect subsequent agent behavior (Xiong et al., 2026). Together, these studies substantially broaden memory evaluation, but evaluation remains largely organized around predefined interaction histories and test queries at fixed evaluation points.

Reliability and security of agent memory. Persistent memory can introduce reliability and security risks when stored information is malicious, stale, conflicting, or otherwise poorly managed (Patlan et al., 2025; Wei et al., 2025). Memory-poisoning and injection studies show that adversarial records can be introduced or activated to influence later agent behavior (Chen et al., 2024; Dong et al., 2025), while benchmark and empirical studies expose difficulties involving knowledge updates, forgetting, experience reuse, and error propagation (Wu et al., 2025; Hu et al., 2026; Xiong et al., 2026). Existing work therefore establishes that memory reliability depends on both stored content and its subsequent use, but primarily studies predefined attacks, benchmark cases, or controlled memory-management operations.

Fuzzing and testing of LLM-based systems. LLMs have been incorporated into fuzzing both as generators of test inputs for software systems and as systems under test through prompt mutation (Deng et al., 2023; Xia et al., 2024; Yu et al., 2024; Shao et al., 2026). Retrieval-aware testing has further introduced coverage criteria and metamorphic mutations to explore RAG behavior beyond fixed queries (Kim et al., 2026b;a). These studies demonstrate the value of coverage-guided and mutation-based testing, but primarily target software inputs, prompt behavior, or document-level retrieval systems rather than persistent agent memory in which both queries and accumulated memory states evolve across interactions.

3 Categorization of Memory-Use Failures

Persistent memory affects an agent through the information that is selected from memory and used to handle a new request. We use $U(q, M)$ to denote the observable memory-use behavior of an agent when processing query q with memory state M. Depending on the system, $U(q, M)$ may include the memories retrieved for the query, their ranking, or the memory-dependent response produced by the agent. We do not assume that a memory-use failure is caused by any particular internal component.

We categorize memory-use failures by changing one of the two inputs to $U(q, M)$ while keeping the other fixed. This gives two categories:

I. Query-related failures occur when the memory state remains fixed, but memory use does not correctly follow an adaptation from q to q'. The expected behavior depends on the relation between the two queries. For example, if q' is a paraphrase of q, the same relevant memory should continue to support the request; if q' asks for unsupported information, the agent should not rely on unrelated memory. We write this requirement as $\boxed{\mathcal{C}_Q(U(q, M), U(q', M); q, q')}$, where $\mathcal{C}_Q$ specifies the relation expected from the query change, such as preserving memory use under equivalent queries or avoiding unsupported memory use for an unanswerable query. A query-related failure occurs when this condition is violated.

II. Memory-state failures occur when the query remains fixed, but memory use does not correctly follow a controlled change from M to M'. A relevant update, deletion, or addition should be reflected when the agent later uses memory, while an unrelated change should not disturb the information used for the same request. We write this requirement as $\boxed{\mathcal{C}_M(U(q, M), U(q, M'); M, M')}$, where $\mathcal{C}_M$ specifies the relation expected from the memory change, such as preferring the updated value after a replacement or no longer using an entry after it has been deleted. A memory-state failure occurs when this condition is violated.

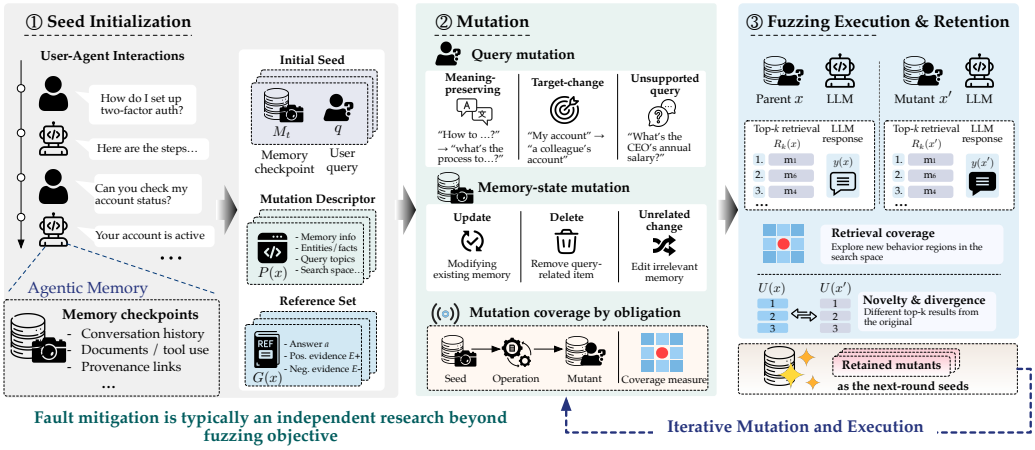


Figure 2: Overview of U-FUZZ. U-FUZZ builds seeds and references from memory checkpoints, generates validated query and memory-state mutations, and executes each parent-mutant pair to detect memory-use failures. Retrieval coverage and divergence guide later fuzzing rounds, while failure labels remain outside the fuzzing search.

This categorization is based on the change that triggers the failure, thus agnostic to how the memory system is integrated to LLM agents. It therefore applies to systems with explicit retrieval interfaces as well as systems whose memory use mechanism is hidden. The mutation operators in Section 4 instantiate these two categories with concrete query and memory changes.

4 METHOD

We develop U-FUZZ, a fuzzing framework for discovering the memory-use failures defined in §3. While iterative mutation and execution are standard in fuzzing (Xia et al., 2024; Deng et al., 2023; Shao et al., 2026), U-FUZZ makes persistent memory use as the fuzzing target by treating both the query and evolving memory state as first-class inputs. As outlined in Figure 2, U-FUZZ first constructs seeds from memory checkpoints (§4.1). It then generates and validates query or memory-state mutations under an explicit mutation-coverage requirement (§4.2). Finally, it executes each valid parent–mutant pair, checks whether the mutation exposes a memory-use failure, and retains useful mutants for later rounds (§4.3).

4.1 SEED INITIALIZATION

U-FUZZ starts from checkpoints taken after an agent has accumulated memory through normal interactions. Let $H_{1:t} = \langle h_1, \ldots, h_t \rangle$ denote the interaction history up to time t, and let M_t denote the resulting memory state. If the system supports state copying, U-FUZZ creates isolated copies of M_t. Otherwise, it recreates the same state in a separate session by replaying $H_{1:t}$. The original task and its memory are left unchanged.

Algorithm 1: Seed Construction

Input: Task history H; checkpoint times $\mathcal{T}$; query sets $\{\mathcal{Q}_t\}$; provenance Z
Output: Seeds $\mathcal{S}_0$; descriptors P; references G
$\mathcal{S}_0 \leftarrow \emptyset$;
foreach $t \in \mathcal{T}$ **do**
  $M_t \leftarrow \textsc{Checkpoint}(H_{1:t})$;
  foreach $q \in \mathcal{Q}_t$ **do**
    $x \leftarrow (M_t, q)$;
    $P(x) \leftarrow \textsc{BuildDescriptor}(q, H_{1:t}, Z)$;
    $G(x) \leftarrow \textsc{BuildReference}(q, H_{1:t}, Z)$;
    $\mathcal{S}_0 \leftarrow \mathcal{S}_0 \cup \{x\}$;
  end
end
return $\mathcal{S}_0, P, G$;

A seed is $x = (M_t, q)$, where q is a query whose answer may depend on information in M_t. As shown in Algorithm 1, each seed has two separate records. The mutation descriptor $P(x)$ contains only information needed to construct valid mutations, such as query slots, available replacement targets, absent targets, and supported memory operations. $P(x)$ is used during mutation (§4.2) and contains no expected answer. Each seed also has a reference set $G(x) = (a, E^+, E^-)$ for evaluation (§4.3). Here, a is the expected answer, E^+ contains valid evidence from memory, and E^- contains evidence made invalid by a memory operation, such as an update or deletion. By default, $E^- = \varnothing$. We use $a = \perp$ when the memory does not contain information that supports an answer to q.

4.2 MUTATION

Given a seed $x = (M_t, q)$, U-FUZZ changes one input to memory use at a time. A **query mutation** produces $x' = (M_t, q')$, while a **memory-state mutation** produces $x' = (M'_t, q)$. We change only one input in each pair so that an observed difference can be attributed to the query change or the memory-state change. A retained mutant may become a seed in a later round, so different changes can accumulate across rounds.

I. Query mutation: A query mutation keeps M_t fixed. U-FUZZ first decides how q' should differ from q, then constructs a query that satisfies this change. **(I-1) Meaning-preserving mutation.** The requested entity, attribute, time scope, answerability, and expected answer remain unchanged, so $a' = a$. Operators include paraphrasing, shortening removable context, adding nonrestrictive context, and changing syntactic form. **(I-2) Target-changing mutation.** The mutant changes one requested target while preserving the rest of the query. If z is a replaceable entity, attribute, or time scope and z' is another target listed in $P(x)$, then $q' = q[z \leftarrow z']$. We construct $G(x')$ from the provenance of z', while the search does not receive this reference. **(I-3) Unsupported mutation.** The mutant replaces the requested target with $z_\emptyset$, whose absence is recorded in $P(x)$, and therefore $a' = \perp$. To avoid confusing missing memory with parametric knowledge of the base LLM, we use only *memory-exclusive* unsupported targets: synthetic or session-local entities and attributes whose answers are not available from public model knowledge. Their absence is established before execution and is never inferred from a failed retrieval or response.

II. Memory-state mutation: A memory-state mutation keeps q fixed and applies a controlled operation δ to an isolated copy of M_t, producing $M'_t = \delta(M_t)$. If the system exposes memory operations, U-FUZZ uses them directly. If memory changes only through interaction, the required interaction is appended to the history that produced M_t, and M'_t is created in a separate session. Specifically, we design three operations: **(II-1) Update** adds a new value v_{new} for an existing entity-attribute pair. For $v_{\text{old}} \rightarrow v_{\text{new}}$, we set $a' = v_{\text{new}}$ and place the old evidence in E'^{-}. **(II-2) Deletion** removes an existing fact; the deleted evidence is placed in E'^{-}, and $a' = \perp$ if no valid support remains. **(II-3) Unrelated change** modifies information outside the target of q, so $G(x') = G(x)$. Its expected effect is no change in memory use for the fixed query. For mutation types that do not create explicitly invalid evidence, $E'^{-} = \varnothing$.

Detailed construction and validation methods are given in Appendix B.1. Generated mutants are accepted only when the operator-specific conditions can be checked independently of the mutant's retrieval result or final response.

Mutation coverage: The space of natural-language queries is unbounded, so U-FUZZ defines coverage over the finite mutation opportunities available from the current seed corpus. At the beginning of round r, the seed corpus $\mathcal{S}_r$ is fixed. We define a mutation obligation $\omega = (x, o, u)$, which pairs a seed x, an applicable operator o, and one concrete mutation target u. For a query mutation, u is the query element or replacement target being changed; for a memory-state mutation, u identifies the memory entry involved in the operation. Let $\mathcal{U}_o(x)$ denote the valid targets of operator o for seed x. We have:

$$\Omega_r = \{(x, o, u) \mid x \in \mathcal{S}_r,\ u \in \mathcal{U}_o(x),\ \text{pre}_o(x, u) = 1\} \quad \text{Cov}^{(r)}_{\text{mut}} = \frac{|\Omega^{\text{exec}}_r|}{|\Omega_r|} \tag{1}$$

where $\text{pre}_o(x, u)$ specifies whether operator o can be applied to seed x at target u, and Ω^{exec}_r contains obligations that have produced at least one validated and executed mutant.

Each uncovered obligation remains in the generation queue until it produces a valid mutant or reaches a fixed generation-attempt limit A. Extra variants are generated only after the uncovered obligations have been tried. Thus, if every obligation yields a valid mutant within A attempts and the round has enough execution budget for one such mutant per obligation, then $\text{Cov}^{(r)}_{\text{mut}} = 1$. This guarantee applies only to the operators and targets represented in Ω_r. Retained mutants are added to $\mathcal{S}_{r+1}$, where they may create new obligations.

4.3 FUZZING EXECUTION AND EVALUATION

For each validated pair (x, x'), both executions start from independent copies of the parent checkpoint M_t. For a query mutation, M_t is copied twice and the two queries are evaluated separately. For a memory-state mutation, one copy remains at M_t, while δ is applied to the other to produce M'_t. If

copying is unavailable, the corresponding checkpoint is recreated in separate sessions. This prevents one test query, including any memory write it may trigger, from changing the other execution.

Let $U(x)$ denote the observed memory-use behavior. For a retrieval-visible system, we record $U(x) = (R_k(x), y(x))$, where $R_k(x) = [m_1, \ldots, m_k]$ is the ranked top-k retrieved memory and $y(x)$ is the final response. For a retrieval-hidden system, only $y(x)$ is available.

Algorithm 2: Execute and Retain

Input: Parent x; validated mutant x'; operator o
Output: Failure result F_o; retain flag b
$U(x), U(x') \leftarrow$ IsolatedExec(x, x');
// Evaluation
$F_o \leftarrow$
  CheckFailure$(o, U(x), U(x'), G(x), G(x'))$;
// Fuzzing feedback
if *retrieval is visible* **then**
  $b \leftarrow$ RetrievalRetention(x, x', o);
else
  $b \leftarrow$ ResponseRetention(x');
end
Store F_o separately;
if $b = 1$ **then**
  Add x' to next-round seeds;
end
return F_o, b;

Evaluation of Memory-Use Failure: For retrieval-visible systems, entries in $R_k(x)$ are matched with E^+ and E^- using memory identifiers or provenance. We define $r^+(x) = \min_{m \in E^+} \mathrm{rank}(m, R_k(x))$, and $r^-(x) = \min_{m \in E^-} \mathrm{rank}(m, R_k(x))$ when $E^- \neq \varnothing$; otherwise $r^-(x) = \infty$. A missing item also has rank ∞. In regular agent use, we check $\mathrm{Ans}(y(x), a)$ to determine whether the final response gives the expected answer. In fuzzing, for each mutation operator o, a memory-use failure occurs when the observed behavior violates an expectation $\mathcal{C}_o$ between the parent and mutant:

$$F_o(x, x') = \mathbf{1}[\neg \mathcal{C}_o\,(U(x), U(x'); G(x), G(x'))] \tag{2}$$

where $F_o = 1$ denotes a memory-use failure. For query mutations, $F_o = 1$ if **(I-1)** a meaning-preserving mutant no longer preserves the expected answer or valid support; **(I-2)** a target-changing mutant does not follow $G(x')$; or **(I-3)** an unsupported query receives a factual answer despite $a' = \perp$. For memory-state mutations, $F_o = 1$ if **(II-1)** updated evidence is missing ($r^+(x') = \infty$) or stale evidence is ranked above it ($r^-(x') < r^+(x')$); **(II-2)** deleted evidence in E'^- is still retrieved or used; or **(II-3)** the answer or valid support changes even though the query-relevant memory remains unchanged.

When retrieval is visible, U-FUZZ can further localize where the failure appears. Missing valid evidence or invalid evidence ranked above it indicates a failure at memory selection. If valid evidence is retrieved but the final response is incorrect, the failure appears after selection. U-FUZZ does not attribute such failures to a specific internal LLM operation, since follow-up diagnosis is outside the fuzzing objective. For systems that hide retrieval details, U-FUZZ reports only the end-to-end memory-use failure.

Seed retention and iteration: Failure detection is kept separate from fuzzing: F_o is stored for evaluation but never used to retain or prioritize seeds. For systems with visible retrieval, U-FUZZ uses three feedback indicators: *coverage* for reaching unseen memory entries, *novelty* for new top-k retrieval patterns, and *divergence* for large parent-mutant ranking changes in meaning-preserving mutations. Coverage and novelty determine retention, while divergence further guides prioritization. For API-based systems with hidden retrieval, U-FUZZ instead retains mutants that produce previously unseen response behavior. At the end of round r, retained mutants are added to $\mathcal{S}_{r+1}$ and used to generate new mutation obligations. More details are given in Appendix B.2.

5 EXPERIMENT

Memory systems: We evaluate on four retrieval-visible memory systems: Mem0 (Chhikara et al., 2025), A-Mem (Xu et al., 2026), Graphiti (Rasmussen et al., 2025), and MemOS (Kang et al., 2025). These systems expose the memories selected for each query and form the default setting. We also separately removes retrieval visibility and evaluates U-FUZZ using only the final response (§5.4).

Benchmarks: We use LoCoMo (Maharana et al., 2024) and LongMemEval-S (Wu et al., 2025), which provide long interaction histories together with memory-dependent queries and references. We construct memory checkpoints and initial seeds from the benchmark interactions following §4.1.

Baselines: We compare U-FUZZ with four baseline strategies: *(i) Random Mutation* defines a uniform obligations across mutants. *(ii) Unguided LLM* uses the LLM itself to generate and select mutants. *(iii) LLM-as-Judge* ranks valid mutants by their estimated likelihood of exposing a fail-

Table 1: Performance under a fixed execution budget B (B=8000 for LoCoMo and B=4000 for LongMemEval-S). Results report mean and standard deviation over three runs. Appendix C also provides some case studies of found memory-use failures.

	Method	Mem0		A-Mem		Graphiti		MemOS	
		UF@B	Cov@B	UF@B	Cov@B	UF@B	Cov@B	UF@B	Cov@B
LoCoMo	Random Mutation	12.7 $\pm$1.5	0.40 $\pm$.02	21.7 $\pm$5.1	0.24 $\pm$.08	18.7 $\pm$3.2	0.27 $\pm$.12	15.0 $\pm$4.6	0.34 $\pm$.10
	Unguided LLM	22.0 $\pm$2.0	0.47 $\pm$.03	29.7 $\pm$3.8	0.39 $\pm$.08	27.3 $\pm$4.0	0.35 $\pm$.10	26.0 $\pm$3.0	0.45 $\pm$.12
	LLM-as-Judge	23.3 $\pm$2.5	0.41 $\pm$.01	35.7 $\pm$2.3	0.38 $\pm$.04	23.3 $\pm$1.5	0.34 $\pm$.02	25.3 $\pm$0.6	0.45 $\pm$.02
	Coverage-Guided	27.7 $\pm$4.0	0.59 $\pm$.03	31.3 $\pm$2.5	0.54 $\pm$.05	30.3 $\pm$3.5	**0.47** $\pm$.04	30.0 $\pm$1.0	0.65 $\pm$.02
	U-Fuzz-Q	33.0 $\pm$2.6	0.57 $\pm$.07	57.0 $\pm$1.0	0.51 $\pm$.03	39.7 $\pm$2.9	0.42 $\pm$.05	41.3 $\pm$2.5	0.57 $\pm$.03
	U-Fuzz-M	27.0 $\pm$2.0	0.54 $\pm$.03	48.3 $\pm$2.1	0.45 $\pm$.06	51.3 $\pm$2.1	0.42 $\pm$.04	27.3 $\pm$0.6	0.54 $\pm$.01
	U-Fuzz	**45.3** $\pm$1.2	**0.62** $\pm$.02	**73.3** $\pm$3.1	**0.55** $\pm$.02	**62.0** $\pm$2.0	0.46 $\pm$.03	**53.7** $\pm$1.2	**0.67** $\pm$.02
LongMemEval-S	Random Mutation	9.3 $\pm$4.9	0.29 $\pm$.07	16.0 $\pm$3.5	0.34 $\pm$.12	17.3 $\pm$4.5	0.25 $\pm$.09	12.7 $\pm$5.9	0.33 $\pm$.08
	Unguided LLM	26.0 $\pm$4.4	0.42 $\pm$.06	32.0 $\pm$3.6	0.42 $\pm$.05	23.7 $\pm$4.6	0.34 $\pm$.08	25.7 $\pm$4.2	0.44 $\pm$.07
	LLM-as-Judge	21.7 $\pm$1.5	0.38 $\pm$.04	39.3 $\pm$1.2	0.39 $\pm$.04	32.0 $\pm$3.0	0.32 $\pm$.03	23.0 $\pm$2.6	0.40 $\pm$.03
	Coverage-Guided	25.0 $\pm$3.5	0.60 $\pm$.02	33.3 $\pm$3.2	0.55 $\pm$.03	30.7 $\pm$3.5	0.46 $\pm$.05	29.0 $\pm$2.6	0.62 $\pm$.02
	U-Fuzz-Q	31.3 $\pm$2.5	0.53 $\pm$.03	48.0 $\pm$2.0	0.53 $\pm$.04	36.7 $\pm$1.5	0.40 $\pm$.02	35.7 $\pm$1.2	0.57 $\pm$.02
	U-Fuzz-M	28.7 $\pm$2.5	0.53 $\pm$.03	45.3 $\pm$3.2	0.48 $\pm$.03	42.7 $\pm$2.9	0.42 $\pm$.01	26.0 $\pm$3.0	0.53 $\pm$.01
	U-Fuzz	**40.3** $\pm$2.5	**0.62** $\pm$.02	**66.3** $\pm$0.6	**0.57** $\pm$.02	**53.0** $\pm$1.0	**0.48** $\pm$.00	**42.7** $\pm$1.2	**0.66** $\pm$.03

ure, without access to references. *(iv) Coverage-Guided* retains and prioritizes mutants using newly reached memory entries only. Other settings (seed, budget) are same across baselines and U-FUZZ.

Metrics. We evaluate fuzzing under an execution budget B by two complementary metrics: *(i) UF@B* counts the unique memory-use failures found within B valid executions. *(ii) Cov@B* measures the fraction of memory entries retrieved at least once.

5.1 RQ1: How effective is U-Fuzz at memory-use failure discovery?

We first compare overall fuzzing effectiveness under a fixed execution budget. Table 1 shows results across all fuzzing methods on both benchmarks and four memory systems, wherein we also consider two adaptations of U-FUZZ: U-FUZZ-Q and U-FUZZ-M uses query-only or memory-state-only mutations, respectively. We have the following observations and insights:

Behavior-guided fuzzing improves failure discovery without reducing the search to coverage alone. Table 1 shows that U-FUZZ consistently discovers more unique memory-use failures than the baseline strategies across benchmarks and memory systems. Coverage-Guided is effective at expanding the set of reached memories, but this does not translate directly into the same level of failure discovery. U-FUZZ maintains comparable or stronger memory coverage while also exploring different behaviors around the memories already reached. This distinction matters because fuzzing methods that reach an entry only ensures that memory is accessible; it does not inspect how reliably the agent uses that entry under different queries and memory states.

Query and memory-state mutations are complementary. The variants of U-FUZZ show complementary behavior. Query mutations are more effective on Mem0, A-Mem, and MemOS, because retrieval in these systems is more directly shaped by query formulation. Memory-state mutations are stronger on Graphiti, where updates and deletions can alter the graph structure and subsequent retrieval paths. This supports our categorization in §3: memory-use failures can be triggered by either query changes or memory-state changes, with their relative importance depending on the backend.

Execution feedback is necessary. LLM-as-Judge is competitive in some settings, but its effectiveness varies across memory systems and benchmarks. This suggests that asking an LLM to predict failure-triggering mutants is not always reliable. In contrast, feedback from the system's actual execution behavior provides a firmer basis for guiding later fuzzing rounds.

5.2 RQ2: How does fuzzing effectiveness evolve with the execution budget?

We next study how U-FUZZ's performance changes with the execution budget. Figure 3 shows the results across memory systems.

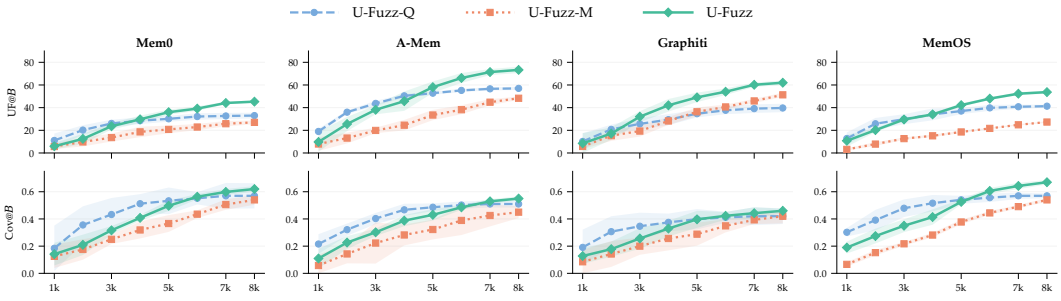


Figure 3: Fuzzing progress as the execution budget (x-axis) increases on LoCoMo. Each column corresponds to one memory system. Shaded regions indicate one standard deviation over three runs.

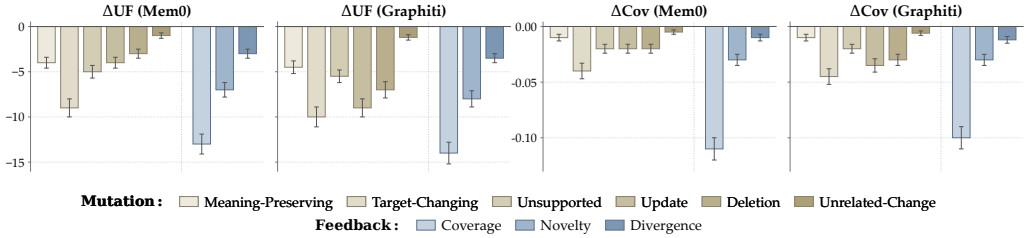


Figure 4: Component ablation on LoCoMo. Each bar reports the change in UF@B or Cov@B after removing one component from the full U-FUZZ. *Mutation* ablations remove one query- or memory-state operator, while *Feedback* ablations remove one search-feedback indicator. Error bars show one standard deviation over three runs.

The two mutation classes show different dynamics. Query-only fuzzing (U-FUZZ-Q) progresses faster at the beginning because it can reuse an existing checkpoint and immediately inspect different query formats. However, it tends to saturate earlier as the available query variations become exhausted. Memory-state fuzzing (U-FUZZ-M) starts more slowly because each mutant requires constructing and validating a new memory state, but it continues to uncover failures that U-FUZZ-Q cannot reach. The two variants therefore complement each other in longitudinal fuzzing runs.

U-FUZZ remains effective as either search direction starts to saturate. The full U-FUZZ combines both mutation classes, so later fuzzing rounds can continue through memory-state changes when query exploration slows, and vice versa. This makes the search more comprehensive over longer budgets and helps avoid early saturation around a single type of memory-use behavior.

5.3 RQ3: How Does Each Component of U-Fuzz Influence to Its Effectiveness?

We ablate U-FUZZ to understand how each individual component influence to it's performance. On the mutation side, we independently remove meaning-preserving (MP), target-changing (TC), unsupported (UNS), update (UPD), deletion (DEL), or unrelated-change (URC) mutations. On the execution side, we independently remove coverage (Cov), retrieval novelty (Nov), or parent-mutant divergence (Div).

Different mutation operators expose different failure cases. Figure 4 shows that removing any mutation operator reduces failure discovery, though the impact varies across backends. Target-changing contributes most among query mutations because it redirects the request to other supported targets and expands the memory being tested. Meaning-preserving and unsupported mutations affect coverage less, but probe failure types that target changes do not capture. Memory-state mutations show stronger backend dependence: update and deletion matter more on Graphiti than on Mem0, suggesting that the two systems respond differently to evolving memory states. Unrelated change has a smaller effect, as expected from its role as a non-interference test rather than a mechanism for reaching new relevant memory.

Execution feedback serves different roles. Coverage helps U-FUZZ reach memory entries that have not been tested before. Retrieval novelty captures new top-k retrieval patterns among entries that may already be covered, while parent–mutant divergence measures how much the ranking changes after a meaning-preserving mutation. The ablation results show that coverage is the main driver of broader exploration, whereas novelty and divergence help U-FUZZ continue finding distinct behaviors within the explored memory space.

5.4 RQ4: DOES U-FUZZ REMAIN EFFECTIVE WHEN RETRIEVAL IS HIDDEN?

The previous experiments assume access to ranked retrieval results. We finally consider an output-only setting in which the memory system executes normally, but its retrieved entries and rankings are hidden from the fuzzer. Thus, U-FUZZ observes $U(x) = y(x)$ rather than $U(x) = (R_k(x), y(x))$. We place Mem0 and Graphiti behind this interface and evaluate them on LoCoMo with GPT-5.5 and Claude Sonnet 4.6 as the LLM backends. U-FUZZ retains the same mutation-obligation schedule, but replaces retrieval-based retention with novel response behavior as described in §4.3. Each configuration uses $B = 2000$ and compares Random Mutation, Unguided LLM, and the output-only version of U-FUZZ. Coverage-Guided is omitted because memory coverage is not observable. Failure confirmation still uses $G(x)$, $G(x')$, and the mutation-specific expectation $\mathcal{C}_o$, but any failure can only be reported end to end.

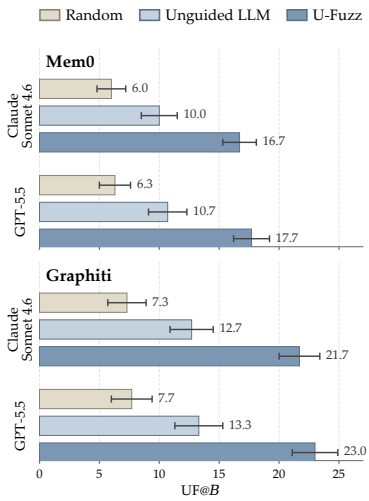


Figure 5: Failure discovery under output-only observability on LoCoMo. Each bar reports UF@B $B = 2000$, wherein we prompt LLMs to output reasoning and responses. Error bars indicate one standard deviation over three runs.

U-FUZZ remains effective with only final responses as feedback. Figure 5 shows a consistent advantage for U-FUZZ across both memory systems and LLM backends, even after retrieval-level feedback is removed. This suggests that the search does not depend entirely on observing which memory entries are retrieved. Mutation obligations still ensure that different query and memory-state changes are exercised systematically, while response novelty provides a weaker but still useful signal for deciding which mutants should continue to later rounds.

Hidden retrieval mainly reduces diagnosis process, not the testability. The output-only setting still allows U-FUZZ to confirm whether a mutation inspects failures of memory-use behavior from the final responses. What it loses is the ability to explain where that failure occurs. Without R_k, the same incorrect response may result from retrieving the wrong memory or from misusing the right one. The results therefore suggest that direct retrieval access is valuable for finer search guidance and localization, but is not a requirement for discovering end-to-end memory-use failures.

6 CONCLUSION

We study a reliability problem in persistent-memory agents that is easy to overlook: even when stored memory is correct, an agent may still use it incorrectly as queries or memory states change. We formulate this problem as fuzzing and introduce U-FUZZ, which systematically mutates both queries and memory states, validates each test case, and uses observed execution behavior to guide iterative exploration without relying on failure labels during search. Across multiple memory systems and benchmarks, U-FUZZ uncovers memory-use failures that simpler testing strategies miss, and remains useful even when retrieval is hidden. More broadly, our results suggest that reliable agent memory should be evaluated not only by what is stored, but also by how that memory is used under changing requests and evolving states. **Ethics Statement:** This work studies the reliability of existing agentic memory systems and does not involve human-subject data collection or the release of new sensitive datasets. Our experiments use established research benchmarks and controlled memory mutations for evaluation purposes, with the goal of identifying reliability issues and supporting more robust memory-enabled agents.

AI USE STATEMENT

In this work, we used generative AI tools only for language polishing, grammar correction, and improving the readability of author-written text. We did not use generative AI tools to generate synthetic datasets; develop theoretical models or conceptual frameworks; formulate mathematical claims; provide critical ingredients for proving mathematical claims; assist in writing proofs; propose or refine hypotheses; design or provide feedback on research methodology or experiments; implement methods; assist with translation; clean or reformat datasets; support qualitative or thematic data analysis; or interpret results. We reviewed all AI-assisted edits and take responsibility for the final content of this work, including all text, claims, and artifacts.

REPRODUCIBILITY STATEMENT

We provide an anonymized implementation of U-FUZZ at `https://anonymous.4open.science/r/U-Fuzz-469C`. The main methodology, including seed construction, mutation, execution, evaluation, and feedback-guided iteration, is specified in Section 4, while the experimental systems, benchmarks, baselines, budgets, and evaluation metrics are described in Section 5. Appendix B provides additional implementation details for mutation construction and validation as well as seed retention and iteration, and Appendix C provides concrete examples of the mutation operators and expected behaviors. Together with the released code, these materials are intended to support reproduction of the experiments reported in this paper.

REFERENCES

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 1032–1043, 2016.

Zhaorun Chen, Zhen Xiang, Chaowei Xiao, Dawn Song, and Bo Li. Agentpoison: Red-teaming llm agents via poisoning memory or knowledge bases. *Advances in Neural Information Processing Systems*, 37:130185–130213, 2024.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 423–435, 2023. doi: 10.1145/3597926.3598067.

Shen Dong, Shaochen Xu, Pengfei He, Yige Li, Jiliang Tang, Tianming Liu, Hui Liu, and Zhen Xiang. A practical memory injection attack against LLM agents. *arXiv preprint arXiv:2503.03704*, 2025.

Shen Dong, Shaochen Xu, Pengfei He, Yige Li, Jiliang Tang, Tianming Liu, Hui Liu, and Zhen Xiang. Memory injection attacks on llm agents via query-only interaction. *Advances in Neural Information Processing Systems*, 38:46697–46731, 2026.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in llm agents via incremental multi-turn interactions. In *International Conference on Learning Representations*, volume 2026, pp. 156259–156291, 2026.

Jiazheng Kang, Mingming Ji, Zhe Zhao, and Ting Bai. Memory os of ai agent. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 25972–25981, 2025.

Jinhan Kim, Samuele Pasini, and Paolo Tonella. When knowledge changes: Metamorphic testing of rag systems with mutations. *arXiv preprint arXiv:2607.26843*, 2026a.

Jinhan Kim, Samuele Pasini, and Paolo Tonella. Testing retrieval-augmented generation systems with chunk coverage. *arXiv preprint arXiv:2607.18155*, 2026b.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024.

Atharv Singh Patlan, Ashwin Hebbar, Pramod Viswanath, and Prateek Mittal. Context manipulation attacks: Web agents are susceptible to corrupted memory. *arXiv preprint arXiv:2506.17318*, 2025.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: a temporal knowledge graph architecture for agent memory. *arXiv preprint arXiv:2501.13956*, 2025.

Yangguang Shao, Jiahao Yu, Hanwen Miao, Gaopeng Gou, Zhen Li, and Junzheng Shi. Promptfuzz: Harnessing fuzzing techniques for robust testing of prompt injection in llms. *IEEE Transactions on Information Forensics and Security*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Qianshan Wei, Tengchao Yang, Yaochen Wang, Xinfeng Li, Lijun Li, Zhenfei Yin, Yi Zhan, Thorsten Holz, Zhiqiang Lin, and XiaoFeng Wang. A-memguard: A proactive defense framework for llm-based agent memory. *arXiv preprint arXiv:2510.02373*, 2025.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 86809–86836, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/d813d324dbf0598bbdc9c8e79740ed01-Paper-Conference.pdf.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024. doi: 10.1145/3597503.3639121.

Zidi Xiong, Yuping Lin, Wenya Xie, Pengfei He, Zirui Liu, Jiliang Tang, Himabindu Lakkaraju, and Zhen Xiang. How memory management impacts llm agents: An empirical study of experience-following behavior. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 623–645, 2026.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026.

Jiahao Yu, Xingwei Lin, Zheng Yu, and Xinyu Xing. LLM-Fuzzer: Scaling assessment of large language model jailbreaks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 4657–4674, 2024.

Boqin Yuan, Yue Su, and Kun Yao. Diagnosing retrieval vs. utilization bottlenecks in LLM agent memory. In *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems*, 2026. URL https://openreview.net/forum?id=cxYbqAtBIz.

A Limitations and Scope

Our evaluation focuses on representative persistent-memory systems, benchmarks, and the query- and memory-state mutations defined in this work. As with fuzzing more generally, the explored test space is not exhaustive, and the specific failures discovered may vary with memory backend, model, and system configuration. In addition, U-Fuzz is designed to discover and localize memory-use failures rather than diagnose their internal causes or prescribe mitigation strategies. Extending the framework to additional memory architectures, mutation operators, and deployment settings is a natural direction for future work.

B TECHNICAL IMPLEMENTATION DETAILS

B.1 MUTATION CONSTRUCTION AND VALIDATION

Given a seed $x = (M_t, q)$ and a mutation obligation $\omega = (x, o, u)$, U-FUZZ constructs one candidate mutant according to operator o and the fixed mutation target u. The information needed for mutation comes from the mutation descriptor $P(x)$ and checkpoint provenance, including the query fields, available replacement targets, absent targets, and memory operations supported by the backend. The mutation target and any new value are fixed before execution. When a query requires free-form rewriting, an LLM may be used to realize the intended change in natural language, but it does not choose the mutation target or determine whether the candidate is valid. Each candidate is validated before execution using the parent query, $P(x)$, checkpoint provenance, and observable results of memory operations. The validator never uses the mutant retrieval $R_k(x')$, response $y(x')$, or failure result $F_o(x, x')$.

Meaning-preserving query mutation. For a meaning-preserving mutation, U-FUZZ keeps M_t fixed and rewrites q while preserving its requested entity, attribute, temporal scope, answerability, and other answer-affecting constraints. We implement the four transformations used in the main text: paraphrasing, shortening removable context, adding nonrestrictive context, and changing syntactic form. Query fields recorded in $P(x)$ are treated as protected during generation. A removable span can be deleted directly, and template-based syntactic changes are also applied directly when possible; otherwise, the protected fields and the intended transformation are given to an LLM for language realization. The resulting q' is accepted only when the protected fields remain unchanged and no new condition is introduced that could alter the answer. For rewrites whose wording changes substantially, we additionally use a semantic checker as a rejection filter, but its judgment alone is not sufficient to accept the candidate. Since the query meaning is preserved, the evaluator keeps $a' = a$ and the same valid support for the mutant.

Meaning-Preserving Query Mutation

Parent: "Based on the information stored earlier, where does Emma live?"
Mutant: "Where does Emma live?"
The surrounding phrase is removed, while the requested entity and attribute remain unchanged.

Target-changing query mutation. A target-changing mutation also keeps M_t fixed, but changes one requested entity, attribute, or temporal scope. If z is the selected query field, its replacement z' is chosen from the valid replacement targets recorded in $P(x)$, and must correspond to information supported by the same checkpoint. When the target appears as a replaceable span, U-FUZZ applies $q' = q[z \leftarrow z']$ directly. If this substitution produces an unnatural or ungrammatical query, an LLM rewrites the sentence using the fixed replacement z' while all other query fields remain protected. Validation checks that the designated field changes to z' and that the remaining query fields are unchanged. Once the replacement is fixed, the evaluator constructs $G(x')$ from its checkpoint provenance; this reference is not exposed to the fuzzing search.

Target-Changing Query Mutation

Suppose M_t contains supported residence information for both Emma and Bob.
Parent: "Where does Emma live?"
Mutant: "Where does Bob live?"
Only the requested entity changes, and the reference for Bob is constructed from the same checkpoint.

Unsupported query mutation. For an unsupported mutation, U-FUZZ replaces one requested target with an absent target $z_\emptyset$ recorded in $P(x)$. The replacement must be type-compatible with the original query so that the mutant remains a natural request, and we restrict these mutations to memory-exclusive targets whose answers concern synthetic or session-local information rather than public facts that may already be known by the base LLM. As with target-changing mutations, direct slot replacement is used when possible and an LLM is used only to produce a grammatical realization when needed. Validation checks the checkpoint provenance before execution and accepts the candidate only when no memory record supports the requested entity-attribute-scope combination. Absence is never inferred from a failed retrieval or an incorrect response. For an accepted mutant, the evaluator sets $a' = \perp$.

Unsupported Query Mutation

Suppose the session contains information about the synthetic project *Orion*, including its owner and current status, but no budget code.
Parent: "Who owns Project Orion?"
Mutant: "What is Project Orion's budget code?"
The mutant remains a valid question about the same session-local entity, but the requested information is absent from M_t.

Update memory-state mutation. An update keeps q fixed and changes the value of an existing entity-attribute pair in an isolated copy of M_t. The mutation target u identifies the existing fact and its current value v_{old}; before execution, U-FUZZ chooses a type-compatible value $v_{\text{new}} \neq v_{\text{old}}$. If the backend provides an explicit update operation, U-FUZZ applies it directly. If updates are represented through later interactions, U-FUZZ appends a new interaction expressing v_{new} to the history that produced M_t and lets the memory system process the update normally, producing M'_t. Validation checks that the new value concerns the same entity and attribute and that the backend records it as the later or active value according to its own update semantics. Physical removal of v_{old} is not required, since some systems retain historical records. For evaluation, a' is changed to v_{new}, while evidence for v_{old} is placed in E'^{-}.

Update Memory-State Mutation

Parent memory: "Emma lives in Chicago."
Update: "Emma now lives in Seattle."
The query remains "Where does Emma live?" The updated state M'_t should treat Seattle as the current value, while the earlier Chicago record may remain as history.

Deletion memory-state mutation. A deletion keeps q fixed and removes a selected fact from an isolated copy of M_t. The target u is identified from checkpoint provenance and, when available, by its backend memory identifier. U-FUZZ invokes the backend's native delete or forget operation when one exists; if deletion is expressed through the system's normal interaction interface, the corresponding operation is issued in a separate session. When several records support the same fact, the mutation obligation specifies which support is removed, and all supporting records are removed when the intended mutant requires the fact to become unsupported. Validation requires that the target exists before the operation and is removed or marked inactive afterward; when the backend returns an explicit operation status, the operation must also report success. The deleted evidence is placed in E'^{-}, and the evaluator sets $a' = \perp$ when no valid support for the answer remains.

Deletion Memory-State Mutation

Parent memory: "Emma lives in Chicago."
Deletion: remove the stored residence record for Emma.
If no other residence record remains, the fixed query "Where does Emma live?" has $a' = \perp$ in the mutated state.

Unrelated memory-state mutation. An unrelated change keeps q and its supporting memory unchanged while modifying another part of the memory state. Using checkpoint provenance, U-FUZZ selects an entity-attribute pair that is outside the target of q, or creates a new fact that is unrelated to the requested information, and applies the change through the backend's normal write or update operation. Validation checks that the modified record does not share the query target and that the evidence supporting the fixed query remains unchanged between M_t and M'_t. Because the information needed by q is untouched, the evaluator keeps $G(x') = G(x)$. This mutation acts as a control for whether an irrelevant memory change unexpectedly alters memory use for the same query.

Unrelated Memory-State Mutation

Fixed query: "Where does Emma live?"
Unrelated change: add "Bob prefers skiing."
Emma's residence information is unchanged, so the expected answer and valid support remain the same after the mutation.

General mutation validation. Before execution, every generated mutant is passed through the same validation pipeline. U-FUZZ first checks that the mutant is well formed and that the intended change can be identified unambiguously. For query mutations, we rerun the same field extractor used to build $P(x)$ on the generated query q', and compare the resulting entity, attribute, temporal scope,

and answerability with the operator requirement. A meaning-preserving mutant is kept only when these fields remain unchanged; a target-changing mutant is kept only when the designated target changes to the selected replacement and the remaining fields stay fixed; an unsupported mutant is kept only when the selected memory-exclusive target is still absent from checkpoint provenance. Candidates with malformed text, multiple unintended field changes, ambiguous references, or a target that cannot be verified are discarded.

For memory-state mutations, U-FUZZ validates the requested change by comparing the selected memory item before and after the operation, or by using the backend's operation result when direct state inspection is available. An update must record v_{new} for the same entity-attribute pair targeted by the obligation; a deletion must remove or deactivate the selected support; and an unrelated change must leave the query-relevant evidence unchanged. We write $V_o(x, x') = 1$ when all checks required by operator o pass, and $V_o(x, x') = 0$ otherwise. Only candidates with $V_o(x, x') = 1$ are executed. These checks use the generated query, checkpoint provenance, memory contents, and memory-operation logs, but never the mutant retrieval $R_k(x')$, response $y(x')$, or failure result $F_o(x, x')$.

Validation with hidden memory states. Some tested systems do not expose their stored entries directly. For these systems, U-FUZZ only treats a memory-state mutation as validated when its completion can be checked independently of the test response, such as through a successful memory API call, an operation log, or a memory-management interface showing the resulting state. We do not validate a mutation by asking the same agent whether it remembers the new or deleted fact, since that would use the behavior under test to establish the validity of the test itself. If the requested state change cannot be verified independently, the candidate is not counted as a validated memory-state test.

Across all operators, validation is completed before fuzzing execution. A candidate that fails its operator-specific checks is discarded and its mutation obligation remains uncovered. Thus, whether a mutant is considered valid depends only on the intended query or memory change, not on whether the resulting execution later exposes a memory-use failure.

B.2 Implementation Detail of Seed Retention and Iteration

U-FUZZ uses observed execution behavior to decide which mutants should continue to later fuzzing rounds. Failure results F_o are stored only for evaluation and never affect retention or prioritization. For systems with visible retrieval, U-FUZZ uses three feedback indicators: **retrieval coverage**, **retrieval novelty**, and **retrieval divergence**. The first two determine whether a mutant is retained, while divergence further prioritizes retained meaning-preserving mutants.

Retrieval coverage. Let $R_k(x') = [m_1, \ldots, m_k]$ denote the top-k memories retrieved by mutant x'. We map each memory m to a stable identifier $\rho(m)$, using its entity–attribute pair when available and its backend memory identifier otherwise. The retrieved entries are $\mathrm{Reg}(x') = \{\rho(m) \mid m \in R_k(x')\}$. Let $\mathcal{V}_r$ contain the memory entries reached earlier in round r. We define the retrieval-coverage gain as

$$R_{\text{cov}}(x') = \frac{|\mathrm{Reg}(x') \setminus \mathcal{V}_r|}{\max(1, |\mathrm{Reg}(x')|)}$$

A positive $R_{\text{cov}}(x')$ means that the mutant reaches memory that has not been explored earlier in the round.

Retrieval novelty. Coverage records which entries have been reached, but not how they are selected and ranked. We therefore represent the ordered retrieval result by $\kappa_R(x') = (\rho(m_1), \ldots, \rho(m_k))$. Let $\mathcal{K}_r$ contain the top-k signatures observed earlier in round r, and define $n(x') = 1$ when $\kappa_R(x') \notin \mathcal{K}_r$, and 0 otherwise. Thus, retrieval novelty captures a new top-k retrieval pattern, even when its individual memory entries have already been reached. Following the retention rule in §4.3, a mutant is retained when $R_{\text{cov}}(x') > 0$ or $n(x') = 1$.

Retrieval divergence. For meaning-preserving mutations, U-FUZZ additionally measures how much the retrieval ranking changes from the parent. We use Rank-Biased Overlap (RBO), which places more weight on agreement near the top of two ranked lists. For lists S and T, let $A_d = |S_{1:d} \cap T_{1:d}|/d$. We use $\mathrm{RBO}_p(S, T) = (1 - p) \sum_{d=1}^{k} p^{d-1} A_d + p^k A_k$ with $p = 0.9$, and define

$$R_{\text{div}}(x, x') = 1 - \mathrm{RBO}_{0.9}(R_k(x), R_k(x'))$$

A larger value indicates a larger change near the top of the retrieval ranking. Divergence is used only to prioritize meaning-preserving mutants; it is not an additional retention condition.

Mutation-obligation coverage is handled before this retention priority: uncovered obligations receive generation attempts before extra variants are assigned to obligations that have already been exercised. When multiple retained mutants compete for the remaining execution budget, U-FUZZ combines the three retrieval-feedback indicators above. Let $\mathbb{1}_{\mathrm{mp}}(x') = 1$ when x' is produced by a meaning-preserving mutation and 0 otherwise. We prioritize retained mutants by

$$s(x') = \frac{R_{\mathrm{cov}}(x') + n(x') + \mathbb{1}_{\mathrm{mp}}(x') R_{\mathrm{div}}(x, x')}{2 + \mathbb{1}_{\mathrm{mp}}(x')} \tag{3}$$

which gives equal weight to the available feedback indicators. After each execution, $\mathcal{V}_r$ and $\mathcal{K}_r$ are updated before the next mutant is considered.

Hidden retrieval. When retrieval is hidden, U-FUZZ cannot observe retrieval coverage, retrieval novelty, or retrieval divergence. It therefore retains mutants based on previously unseen response behavior, following §4.3. Each response $y(x')$ is encoded with a fixed sentence encoder and compared with response clusters already observed in the round. If its maximum cosine similarity to the existing clusters is below a fixed threshold τ, it forms a new cluster and the mutant is retained. The encoder and τ are fixed before fuzzing.

Let $\mathcal{R}_r$ denote the mutants retained in round r. The next seed corpus is $\mathcal{S}_{r+1} = \mathcal{S}_r \cup \mathcal{R}_r$. Each retained mutant is then treated as an ordinary seed in the next round: its current memory state takes the role of M_t, and either its query or memory state may be mutated again. Failure results F_o remain separate throughout this process.

C MUTATION CASE STUDIES

We provide concrete examples of the six mutation operators used by U-FUZZ. The query-mutation examples are constructed from benchmark questions and references in LoCoMo (Maharana et al., 2024) and LongMemEval-S (Wu et al., 2025). For memory-state mutations, we start from benchmark-derived memory states and apply the valid state transition defined by the corresponding operator. These examples illustrate the parent–mutant relation and the expected behavior checked by U-FUZZ; they do not assume that every mutant necessarily triggers a failure on every memory backend.

Case Study 1: Meaning-Preserving Query Mutation

A surface reformulation should preserve the same memory use. LoCoMo contains the query *"What are Melanie's pets' names?"*, with the expected answer *Oliver, Luna, Bailey*. U-FUZZ rewrites it as *"What are the names of Melanie's pets?"* while preserving the requested entity, attribute, temporal scope, and answerability. The parent–mutant pair is $(M, q) \rightarrow (M, q')$, and the reference remains unchanged, $G(M, q') = G(M, q)$. A failure is exposed if the reformulation causes the agent to lose the supporting pet memories or return an answer inconsistent with the original benchmark-supported answer.

Case Study 2: Target-Changing Query Mutation

Changing one supported target should redirect memory use to the new target. LoCoMo asks *"When did Melanie go to the museum?"*, whose expected answer is *5 July 2023*. The same conversation also supports *"When did Melanie go to the pottery workshop?"*, with the expected answer *the Friday before 15 July 2023*. U-FUZZ therefore changes the event target from *museum* to *pottery workshop* while preserving the entity and requested temporal relation. Unlike the meaning-preserving case, the mutant has a new reference derived independently from the same checkpoint. A failure is exposed if the system continues to use the museum evidence, returns the museum date, or otherwise fails to switch to the evidence supporting the new target.

Case Study 3: Unsupported Query Mutation

Replacing a supported target with an unsupported target should remove answerability. LongMemEval-S contains the supported query *"Which project did I start first, the Ferrari model or the Japanese Zero fighter plane model?"*, whose expected answer is *the Japanese Zero fighter plane model*. A corresponding abstention case replaces the second target with the *Porsche 991 Turbo S model*, which is not supported as a started project by the available history. This gives the mutation $q[z \leftarrow z_\emptyset]$ with $a' = \perp$.

The lack of support is established from checkpoint provenance before the mutant is executed. A failure is exposed if the agent nevertheless returns a definite memory-backed ordering instead of recognizing that the available memory is insufficient.

Case Study 4: Update Mutation

A later valid value should replace the earlier value used for the same query. A LongMemEval-S knowledge-update instance records a change in the user's tennis schedule at the local park: the earlier state says that the user plays with friends *every week on Sunday*, while the later state says *every other week on Sunday*. U-FUZZ keeps the query *"How often do I play tennis with my friends at the local park?"* fixed and changes only the memory state, $(M, q) \rightarrow (M', q)$. The later interaction is supplied to the tested memory system, which constructs M' through its native update behavior; U-FUZZ does not manually rewrite the resulting memory. After the update, the expected answer is the new frequency, while evidence supporting the weekly schedule becomes stale evidence in E^-. Returning *every week* after the update exposes a stale-memory failure.

Case Study 5: Deletion Mutation

Deleted information should no longer support the fixed query. LoCoMo contains the query *"What did Caroline research?"*, with the expected answer *Adoption agencies*. Starting from a checkpoint in which the corresponding fact is stored, U-FUZZ removes the memory entry supporting that fact while keeping the query fixed: $(M, q) \rightarrow (M', q)$. The deletion is performed through the tested backend's native memory operation and is independently verified before the mutant is executed. If no other valid support remains, the post-deletion reference becomes $a' = \perp$, and the removed evidence is placed in E^-. Retrieving the deleted evidence or continuing to answer *Adoption agencies* after the deletion constitutes a deletion failure.

Case Study 6: Unrelated-Change Mutation

An unrelated memory change should not perturb the result of a fixed query. LoCoMo asks *"What instruments does Melanie play?"*, with the expected answer *clarinet and violin*. The same conversation contains unrelated information about Caroline's adoption process. U-FUZZ introduces a valid write concerning this Caroline-related information while leaving the query about Melanie unchanged. Because the new memory concerns a different entity and attribute, the query reference must remain invariant: $G(M', q) = G(M, q)$. The backend is free to incorporate the new information into its memory state, but the evidence and answer used for Melanie's instruments should remain unaffected. If the unrelated write changes the selected support or alters the final answer, U-FUZZ reports a non-interference failure.