HoMe: Toward Practical Harness Optimization of Memory Systems for AI Assistants

Anonymous authors
Paper under double-blind review

Abstract

The behavior of memory systems for long-running AI assistants is governed by a harness, which consists of the code, prompts, and configurations for constructing, maintaining, and retrieving memory. However, optimizing the harness of a memory system remains largely manual, and existing automated methods restrict either the editable components or the search procedure. In this paper, we introduce HoMe, a pure evaluation-driven framework that iteratively evaluates candidate harnesses and derives each new candidate from any previously evaluated harness by editing any part of it. HoMe combines session-level memory versioning with execution traces to localize failures to the responsible sessions and harness components. It further reduces evaluation cost through smoke tests, candidate gating, adaptive sampling, and the reuse of constructed memory. We validate HoMe with the open-source memory system on two large-scale long-horizon memory benchmarks, LongMemEval-S and BEAM. With only six evaluated candidates per benchmark, HoMe discovers harnesses that improve end-task performance over the originally strong harness while consuming far fewer query-time tokens.

1 Introduction

Long-running AI assistants accumulate conversational and episodic experience through continued interaction, and they depend on a memory system to turn that experience into consistent and personalized behavior across sessions (Packer et al., 2023; Chhikara et al., 2025; Xu et al., 2025; Zhong et al., 2023; Wang & Chen, 2025). In practice, memory is not merely persistent storage but a standalone system built around a language model, exposing interfaces for a target agent to store and retrieve information as needed. Many such systems are file-native, i.e., maintaining memory as files (Zhou et al., 2026). Three components govern the system behavior: the code that implements memory construction, maintenance, and retrieval, along with associated evaluation and recovery mechanisms; the prompts that instruct the language model at each stage; and the configurations that determine the memory schemas, indexes, thresholds, etc. We call this collection of code, prompts, and configurations the *harness* of a memory system.

Different AI assistants often have different demands on memory data management. For example, a medical assistant stores symptom histories and medication details that must be preserved faithfully, whereas a scheduling assistant stores event states that are revised repeatedly and should be superseded rather than accumulated (Wu et al., 2024; 2026). Hence, the harness of a memory system must be optimized before deployment to achieve acceptable results on application-specific evaluations (Yao et al., 2026; Zhang et al., 2026). The same effort is required again after deployment, because growing interaction histories and shifting user behavior expose failures that could not be anticipated beforehand. This process is extremely time-consuming due to the huge search space and is still carried out manually, which raises a timely and important question: how can the harness of a memory system be optimized automatically?

Prior work has largely treated the harness of a memory system as an artifact to be designed. Most efforts propose new architectures and retrieval strategies (Packer et al., 2023; Gutiérrez et al., 2024; Xu et al., 2025; Rasmussen et al., 2025; Pan et al., 2025; Kim et al., 2025; Fang et al., 2025; Sun et al., 2026), or grant the target agent control over its memory operations, such as deciding what to write and when to retrieve (Zhang et al., 2024; Hu et al., 2025). A recent and related line of work focuses on harness optimization of a memory system. EvolveMem (Liu et al., 2026b) opti-

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.

mizes retrieval strategies in an autonomous research manner, MemPro (Liu et al., 2026c) optimizes the memory construction and retrieval workflow, and MStar (Pan et al., 2026) optimizes a broader memory program comprising memory schemas, read and write code, and interaction instructions, using a population-based search algorithm. However, they either optimize only a part of the harness or restrict the harness revisions to predefined actions or a fixed procedure.

In this paper, we move one step further and introduce HoMe, a pure evaluation-driven framework that permits modifying any part of a memory system harness in a flexible way. Our framework is inspired by Meta-Harness (Lee et al., 2026), a general scheme that delegates harness optimization to a code agent, which reads a harness with its evaluation evidence and returns a revised harness. At each iteration, HoMe diagnoses the optimization history, including all the harnesses proposed so far, the memories they constructed, and their evaluation results. It then performs arbitrary edits to propose a new candidate harness and evaluates it accordingly. The loop continues until it reaches the maximum iteration and produces the strongest evaluated candidate.

However, applying such a flexible harness optimization scheme to a memory system for AI assistants is not viable, because the system operates over a long horizon (e.g., exceeding thousands of messages (Tavakoli et al., 2025)), which raises two *challenges*. First, accurate diagnosis is essential yet difficult. A query is answered from memory accumulated over hundreds of preceding sessions, and a low score may originate in how a session was segmented and written, in a later revision that overwrote relevant content, or in the retrieval procedure that selected what was returned. Without locating the responsible part of the harness, there is little guidance on what to change, and iterations are easily wasted on unrelated edits. Second, evaluation is prohibitively expensive. For each candidate harness, evaluation constructs memory over all preceding sessions, answers the evaluation queries, and invokes a judge on every case, where a case comprises one or several queries. Evaluation therefore dominates the runtime of the optimization process.

To address the challenges, HoMe first checkpoints the evolving memory files with Git after every session, which converts an opaque construction history into an inspectable sequence of diffs. For each failed query, HoMe reads the execution trace to determine at which stage the required evidence was lost, whether it was never written, later overwritten, or present in the final memory but not retrieved. It then inspects the diffs of the implicated sessions to confirm the cause, and traces the failure to the responsible part in the harness. We further aggregate the diagnoses across cases to distinguish recurring harness defects from isolated failures. Second, HoMe lowers the evaluation cost along three axes: smoke tests and candidate gating reject broken or clearly inferior harnesses before a formal evaluation is launched, which reduces the number of evaluations; adaptive sampling restricts each evaluation to the most informative cases, reducing the number of cases per evaluation; and the constructed memory will be reused properly if a candidate leaves memory construction behavior unchanged, which reduces the construction cost per case. We also extend version control to the optimization itself: the harness edits and the diagnosis reports of every iteration are committed together, so that each iteration builds on prior attempts rather than rediscovering them.

We implement HoMe on the open-source memory system ReMe (Cao et al., 2026) and evaluate it on the long-horizon benchmarks LongMemEval-S (Wu et al., 2024) and BEAM (Tavakoli et al., 2025). Evaluating only six candidate harnesses per benchmark in several hours, HoMe discovers harnesses that outperform the original one by 2.8 and 5.4 percentage points in evaluation score while reducing query-time token consumption by 59% and 41%, respectively. The harness optimized on BEAM-100K also remains effective on the $10\times$ longer conversations of BEAM-1M, showing it generalizes across memory scales. To summarize, our major contributions are the following.

- We formulate the harness optimization problem of a memory system for long-horizon AI assistants, and introduce HoMe, an evaluation-driven framework that jointly searches the code, prompts, and configurations of the system through free-form harness revisions.

- We identify two challenges that arise in the long-horizon setting, namely accurate diagnosis and evaluation cost, and address them through session-level memory versioning with diff-based diagnosis for the former, and candidate gating, adaptive sampling, and memory reuse for the latter.

- We validate HoMe with the open-source memory system on two long-horizon memory benchmarks, LongMemEval-S and BEAM. By evaluating only six candidate harnesses per benchmark, HoMe identifies harnesses that score higher than the strong original harness while consuming fewer query-time tokens. The discovered harness also generalizes to $10\times$ longer conversations.

2 PRELIMINARIES

We consider a file-backed memory system built around a language model that serves a target agent through two interfaces: memory construction and memory retrieval. Let $\mathcal{S} = (s_1, \ldots, s_T)$ denote a chronologically ordered history of sessions, where each session is an ordered sequence of interaction messages. Let m_t denote the persistent memory state after session s_t, including the memory files and their associated indexes. Starting from an initial state m_0, the system processes the history and answers an evaluation query q as follows:

$$m_t = C_h(m_{t-1}, s_t), t \in [1, T], \quad r_q = R_h(m_T, q), \quad a_q = A_h(q, r_q). \tag{1}$$

Here C denotes the construction process, which covers maintenance operations such as revision and consolidation; R denotes the retrieval process; and r_q denotes the information returned to the target agent A, which produces the answer a_q.

Harness of memory system. The *harness* of a memory system is a triple $h = (c, p, \theta) \in \mathcal{H}$, where c denotes the executable code, p denotes the prompts, and θ denotes the configuration values such as memory schemas, indexing options, and thresholds. These components jointly determine the construction and retrieval processes, i.e., C_h and R_h, including their language model calls, internal control flow, and recovery mechanisms. The search space $\mathcal{H}$ consists of the harnesses that preserve the interfaces in Eq. 1, whereas the language model remains fixed and lies outside h.

Harness evaluation. Let $\mathcal{D} = \{x_i\}_{i=1}^N$ denote the set of application-specific evaluation cases. Each case $x_i = (\mathcal{S}_i, \mathcal{Q}_i, \mathcal{G}_i)$ comprises a session history $\mathcal{S}_i = (s_{i,1}, \ldots, s_{i,T_i})$, a nonempty set of queries $\mathcal{Q}_i$ grounded in that history, and a set of ground-truth answers. Applying h to case x_i according to Eq. 1 yields an answer $a_{i,q}^h$ for every $q \in \mathcal{Q}_i$, which a fixed external judge J scores as $y_{i,q}(h) = J(q, a_{i,q}^h, g_{i,q}) \in [0, 1]$. Evaluating a candidate harness h_b on $\mathcal{D}$ produces:

$$(\mathbf{y}_b, \mathbf{e}_b, \kappa_b) = \mathrm{Eval}(h_b; \mathcal{D}), \tag{2}$$

where $\mathbf{y}_b$ collects the query-level scores; $\mathbf{e}_b$ is the evaluation trajectory comprising the query execution logs and the intermediate memory states; and κ_b records query-stage token consumption, i.e., the average number of tokens consumed per query for retrieval and answer generation.

Harness optimization. In this paper, we study the iterative optimization of the harness of a memory system. Starting from an initial harness h_0, each iteration b uses the accumulated evaluation history $\mathcal{O}_b = \big((h_j, \mathbf{y}_j, \mathbf{e}_j, \kappa_j)\big)_{j=0}^b$ and the evaluation data $\mathcal{D}$ to propose the next candidate:

$$h_{b+1} = \Pi(\mathcal{O}_b, \mathcal{D}), \tag{3}$$

where Π denotes the candidate-generation procedure. Under a budget of B iterations, the optimization finally returns the harness $h^* \in \{h_0, \ldots, h_B\}$ with the highest evaluation score under the benchmark's scoring rule. It also favors harnesses with low query-stage token consumption κ and prevents gains achieved merely by retrieving more context or increasing reasoning and tool calls.

3 DIAGNOSABLE AND EFFICIENT HARNESS OPTIMIZATION

3.1 THE HOME FRAMEWORK

HoMe is an evaluation-driven framework that optimizes the harness of a file-native memory system. As shown in Figure 1, it consists of three major components. The *Developer Agents* take the role that human developers play in manual harness optimization: they diagnose the failures of evaluated candidates and edit a harness to produce a new candidate. The *Experiment Manager* records all candidates and their evaluation results, controls the Developer Agents' access to tools, and decides how each candidate is evaluated. The *Evaluation Environment* executes evaluations in isolated sandboxes. In each iteration, the Developer Agents submit a candidate, the Experiment Manager screens it and plans its evaluation, the Evaluation Environment evaluates it, and the Experiment Manager records the results for subsequent iterations. In HoMe, the Developer Agents and the Experiment Manager jointly realize Π in Eq. 3, and the Evaluation Environment computes Eval in Eq. 2.

Developer agents. The Developer Agents consist of an Optimizer Agent that generates candidates and a Diagnoser Agent that attributes failures to memory construction, retrieval, or answering (Section 3.2). All candidates originate from the initial harness h_0, which is evaluated on the full $\mathcal{D}$ and

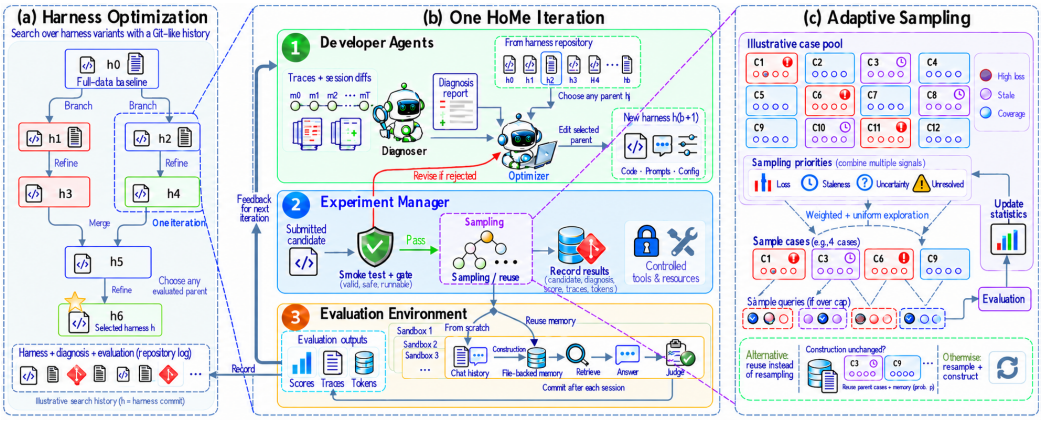

Figure 1: Overview of HoMe. The Developer Agents diagnose failures and edit the harness to generate candidates. The Experiment Manager records candidates and results, controls tool access, and plans evaluations. The Evaluation Environment evaluates candidates in isolated sandboxes.

stored as the root commit of a Git repository maintained by the Experiment Manager. We call this repository the *harness repository*, and each of its branches tests one optimization hypothesis. In each iteration, the Diagnoser Agent reads the optimization history $\mathcal{O}_b$ together with the diagnoses attached to earlier commits and then diagnoses the failures of the most recently evaluated candidate h_b. Based on these diagnoses, the Optimizer Agent selects an evaluated candidate h_j with $j \leq b$ as the *parent* of the next candidate. The choice of parent determines how the search proceeds: extending the latest candidate on a branch refines the hypothesis of that branch; starting a new branch pursues a different hypothesis and may begin from an earlier candidate when the current hypothesis fails to improve performance; and merging two branches combines changes that address complementary failures. Unlike prior work (Lee et al., 2026; Pan et al., 2026) that decides which harness to edit with a fixed rule, HoMe leaves this choice to the Optimizer Agent.

Experiment manager. The Experiment Manager maintains the harness repository, controls the evaluation of each candidate, and provides the Developer Agents with a unified tool interface for code inspection within a permission boundary. A candidate is admitted to evaluation only if it passes a smoke test (Section 3.3). A rejected candidate is returned to the Optimizer Agent with the reason for rejection and consumes no budget. To reduce the evaluation cost for an admitted candidate h_{b+1}, the Experiment Manager judiciously samples an evaluation subset $\mathcal{D}_{b+1} \subseteq \mathcal{D}$ and determines whether the memory constructed for the parent can be reused. It then evaluates h_{b+1} on $\mathcal{D}_{b+1}$ and returns the evaluation record $(\mathbf{y}_{b+1}, \mathbf{e}_{b+1}, \kappa_{b+1})$. We refer to this step as a *formal evaluation*, which consumes one unit of the budget B. Because each commit records its parent, the diagnosis that guided its creation, and its evaluation trajectory, the repository provides complete provenance for every candidate and allows each iteration to build on earlier attempts instead of rediscovering them.

Evaluation environment. The Evaluation Environment executes smoke tests and formal evaluations. In a formal evaluation of h_{b+1}, each case $x_i \in \mathcal{D}_{b+1}$ runs in an isolated Docker sandbox that performs memory construction, retrieval, answering, and scoring with the fixed judge J. All cases run concurrently, and a pool of warm sandboxes reduces startup overhead. During construction, the sandbox commits the memory files after every session and thus produces the memory repository of x_i. When the Experiment Manager permits memory reuse for x_i, the sandbox restores the memory constructed for the parent and runs only retrieval, answering, and scoring.

3.2 Long-Horizon Failure Diagnosis

In each iteration, the Diagnoser Agent analyzes the failed cases of the most recently evaluated candidate h_b, where a case fails if any of its queries receives a low score. For each failed case, it identifies the cause of the failure and the responsible part of the harness. However, a low evaluation score does not reveal the cause. The required information may never have been written to memory, may have been corrupted or deleted by later maintenance, may be present in memory but not retrieved, or

may be retrieved but used incorrectly during answering. A direct diagnosis must therefore examine the *evaluation trajectory* of the case, which consists of the query execution logs and the intermediate memory states (see Eq. 1). The query execution logs are typically short, but the intermediate memory states span many processed sessions. These states are expensive to examine in full and can far exceed the context window of the Diagnoser Agent. The difficulty is compounded because an evaluation often contains many failed cases. HoMe addresses these difficulties with three designs.

Session-level memory versioning. This design makes the intermediate memory states compact and inspectable on demand. During construction, the Evaluation Environment commits the memory files of each case x_i to a Git repository after every session. The commits of this *memory repository* are checkpoints of the intermediate memory states $m_{i,t}$, and the diff between two consecutive checkpoints records exactly what one session changed. An agent can therefore retrieve any intermediate state or any single-session change without reading the entire history.

Trajectory-guided attribution. The Diagnoser Agent diagnoses each failed case by tracing its evaluation trajectory backward from each low-scoring query q. It first checks whether the retrieved information r_q and the final memory state m_{i,T_i} contain the information required by q. If r_q contains the information, the failure lies in answering. If only m_{i,T_i} contains it, the failure lies in retrieval. Otherwise, the failure lies in construction. For a construction failure, the agent locates the sessions $s_{i,t}$ that contain the required information and inspects the diffs between $m_{i,t-1}$ and $m_{i,t}$. These diffs show whether the information was ever written to memory. If it was, the agent follows the later change history of the files that store it and identifies the session whose maintenance corrupted or deleted it. Finally, the agent summarizes the findings in a compact *case diagnosis*, which records the cause of the failure, the session where it originates, and the responsible part of the harness.

Parallel case diagnosis. The Diagnoser Agent follows a map-reduce paradigm to diagnose multiple failed cases efficiently. In the map phase, it diagnoses all failed cases in parallel through the above attribution. Each case is diagnosed in a separate context and yields a compact case diagnosis. The Diagnoser Agent reads only these case diagnoses, so its context grows with the number of failed cases rather than with the length of their trajectories. In the reduce phase, it groups the case diagnoses by cause and separates recurring failure patterns from isolated failures that may stem from evaluation noise. The resulting *diagnosis report* pairs each recurring pattern with the responsible part of the harness and guides the Optimizer Agent in selecting the next parent and editing it.

3.3 Efficient Evaluation for Candidate Harness

The overall evaluation cost depends on three factors: the number of formal evaluations, the number of cases per evaluation, and the average cost per case. HoMe reduces these factors through smoke test and candidate gating, adaptive case and query sampling, and construction-aware memory reuse.

Smoke test and candidate gating. HoMe screens a candidate harness with two checks that cost far less than a formal evaluation. First, it runs a smoke test that executes the candidate on real cases deterministically truncated to a few sessions and queries. This rejects candidates with implementation failures such as import errors, runtime exceptions, and empty outputs. Second, we develop a gate that decides whether a candidate may enter formal evaluation. We observe that an unconstrained optimizer initially proposes many candidates that edit only prompts or configurations, because such edits are cheap to produce. However, the effect of these edits depends on code c, which determines the memory representation and the construction and retrieval procedures. Because later code changes often invalidate this effect, evaluating them before the code stabilizes wastes budget. Hence, we devote the first $B/2$ budget units to an exploration stage and the rest to a refinement stage. During exploration, the gate admits candidates whose edits change the memory representation or the construction and retrieval procedures. During refinement, the gate is lifted, allowing candidates that modify only prompts or configurations as well as those that continue to modify code.

Adaptive case and query sampling. To reduce the cost of each formal evaluation, HoMe restricts it to a subset $\mathcal{D}_{b+1} \subseteq \mathcal{D}$ selected by an adaptive sampler. The sampler treats each case and each query as a *sampling unit*, indexed by v. Intuitively, recently failed or persistently low-scoring units reveal the limitations of the current harness more often than units that consistently pass. However, evaluation is stochastic, and a harness change can cause a previously correct unit to regress. The sampler therefore prioritizes high-loss units while preserving coverage of the remaining units.

The evaluation of h_0 on the full $\mathcal{D}$ initializes the statistics of every unit. After the formal evaluation of h_b, the sampler updates the loss of each evaluated unit v by an exponential moving average:

$$\bar{\ell}_v^{(b)} = \alpha\big(1 - y_v^{(b)}\big) + (1-\alpha)\,\bar{\ell}_v^{(b-1)}, \tag{4}$$

where $y_v^{(b)} \in [0,1]$ is the score of v under h_b and $\alpha \in [0,1]$ weights new observations. For a case, $y_v^{(b)}$ is the mean score of its evaluated queries. A larger α reacts faster to new failures but is more sensitive to evaluation noise. The sampler also increments the count n_v of valid observations of v and records b as the most recent iteration in which v was evaluated. For each unit v in an eligible set $\mathcal{V}$, the sampler then computes a priority r_v and a sampling probability p_v:

$$r_v = \bar{\ell}_v + \lambda_s \log(1+\Delta_v) + \frac{\lambda_n}{\sqrt{n_v+1}} + \lambda_u u_v, \quad p_v = (1-\epsilon)\,\frac{\exp(r_v)}{\sum_{v' \in \mathcal{V}} \exp(r_{v'})} + \frac{\epsilon}{|\mathcal{V}|}. \tag{5}$$

Here, Δ_v is the number of iterations since v was last evaluated, and u_v measures how much of its past error has not yet been rechecked. The weights λ_s, λ_n, and λ_u control the contributions of staleness, observation uncertainty, and unresolved errors, respectively. The constant $\epsilon \in [0,1]$ reserves probability mass for uniform exploration. The loss and unresolved-error terms focus sampling on failures, whereas the staleness and uncertainty terms and the uniform component preserve coverage. A larger ϵ broadens coverage but weakens the focus on failures. The sampler selects units at two levels. It first sets $\mathcal{V} = \mathcal{D}$ and draws cases according to p_v. When the selected cases contain more queries than the per-iteration query cap, it sets $\mathcal{V}$ to these queries and applies the same rule. After the budget is exhausted, the Optimizer Agent uses the evaluation history to select the candidate it judges most likely to achieve the highest score on the full data set while satisfying the token constraint.

Memory reuse. Memory construction dominates the cost of evaluating a case. However, many edits leave construction unchanged, such as edits confined to retrieval. Consider a candidate $h = h_{b+1}$ with parent h'. HoMe inspects the diff between h and h' and determines whether it changes any code, prompt, or configuration used during construction. If it does not, the two candidates have equivalent construction processes C_h and $C_{h'}$. When this holds, with probability ρ, HoMe reuses the evaluation of h': instead of sampling, it directly adopts all cases and their corresponding memory states m_{i,T_i} from h''s evaluation, and runs only retrieval, answering, and scoring. Otherwise, it resamples and constructs from scratch. This probabilistic reuse avoids tying descendant evaluations to a single stochastic construction outcome while enabling broader case coverage.

3.4 Implementation

We implement HoMe on ReMe (Cao et al., 2026), an open-source file-native memory system whose original harness already performs strongly (see Table 1). Improving on this harness therefore provides a stringent test of HoMe. During construction, ReMe extracts information from each session, revises existing content, and writes memory files. In the query stage, it combines retrieval and answering into a single agentic process, in which an Answer Agent repeatedly invokes hybrid search and file-reading tools over the memory files before generating the answer. Inspired by the Everything-is-a-Plugin design (DeepSeek-AI, 2026), we restructure ReMe into a shared runtime and application-specific plugins. The runtime provides infrastructure unrelated to memory management, such as task execution, model access, and storage. Each plugin instantiates an application-specific harness $h = (c, p, \theta)$ of the memory system. This separation has two advantages. First, it enables an accurate evaluation of HoMe. The Evaluation Environment runs the same runtime with the candidate plugin, and performance differences on an application thus reflect changes to the harness rather than to the infrastructure. Second, it simplifies evaluation on new applications. Because the runtime instantiates and dispatches registered plugins according to a job configuration, supporting a new application requires only a new plugin. Various memory designs can be easily implemented by replacing the modules, and HoMe can optimize the resulting harnesses without modification.

Table 1: Overall query-stage results and efficiency. Token counts are averages per query.

Dataset	Method	Score	Total Tok.	In Tok.	Out Tok.	LLM Calls
LongMemEval-S	A-MEM	0.740	19.8K	19.8K	$<$ 0.1K	2.0
	SimpleMem	0.728	37.7K	28.7K	9.0K	8.1
	ReMe	0.888	43.9K	43.2K	0.8K	4.7
	HoMe	**0.916**	17.9K	17.2K	0.6K	3.0
BEAM	A-MEM	0.642	30.3K	28.4K	1.9K	2.0
	SimpleMem	0.440	34.8K	25.2K	9.6K	6.8
	ReMe	0.650	101.8K	100.0K	1.8K	4.5
	HoMe	**0.704**	60.4K	57.8K	2.6K	6.7

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Benchmarks. We evaluate on two long-term memory benchmarks: LongMemEval-S[1] (Wu et al., 2024) and BEAM (Tavakoli et al., 2025). LongMemEval-S contains 500 cases, each with approximately 115K tokens of session history and one query, spanning six question types including single-session understanding and multi-session reasoning. All variants of BEAM share ten question types. Its 100K and 1M variants contain 20 and 35 cases, respectively, with 20 queries per case. Their histories average 131K and 1.04M tokens, respectively, and the longest BEAM-1M history reaches 1.52M tokens. HoMe searches only on training data, while all methods are evaluated on test data. We randomly split LongMemEval-S into equally sized training and test sets; for BEAM, we optimize the harness on the Beam100K and test it on the Beam1M to assess transferability of scales.

Baselines and evaluation. We compare against A-MEM (Xu et al., 2025), SimpleMem (Liu et al., 2026a), and ReMe (Cao et al., 2026). We manually tuned ReMe for each benchmark as the initial harness to test whether HoMe can make further improvements over human optimization. An LLM judge (Zheng et al., 2023) scores answers under each benchmark's official protocol. We also report query-stage tokens, but not capture KV-cache savings because API providers do not expose cache hit rates. ReMe and HoMe use agentic query loops that append to an existing context, enabling greater prefix reuse across calls than the largely independent calls in A-MEM and SimpleMem.

Hyperparameters. We conduct a search over the key hyperparameters of SimpleMem and A-MEM. For SimpleMem, we use $(K_{\text{semantic}}, K_{\text{keyword}}, K_{\text{structured}}) = (40, 10, 10)$ on LongMemEval-S and $(35, 8, 8)$ on BEAM, with at most three reflection rounds on either benchmark. For A-MEM, we set the retrieval count K to 16 and 10, respectively. For HoMe, we set the formal evaluation budget to $B = 6$. Each round samples 28 cases on LongMemEval-S, and 10 cases with 100 questions on BEAM. Concurrency is set to 28. We set the α to 0.3 and ϵ to 0.2 in Eq. 4 and 5. Memory reuse probability ρ is set to 0.6. Across all methods, memory construction uses `qwen3.6-flash`. All other LLM API calls including answering, judging, and the Developer Agents in HoMe use `qwen3.7-max`. Embeddings are generated with `Qwen3-Embedding-0.6B`.

4.2 MAIN RESULTS

4.2.1 OPTIMIZATION EFFECTIVENESS

From the initial harnesses to the Optimizer's final harnesses, the LongMemEval-S and BEAM runs make 1,379 and 891 line edits, respectively, spanning code c, prompts p, and configuration θ, as detailed in Tables 6 and 7. Table 1 shows that HoMe achieves the highest answer score on both benchmarks. Relative to the initial ReMe harness, its LongMemEval-S score rises from 0.888 to 0.916 (+2.8 percentage points), while average query-stage tokens fall from 43.9K to 17.9K (a 59% reduction). LLM calls fall from 4.7 to 3.0 per query, consistent with the optimized retrieval locating relevant evidence with fewer search rounds. On BEAM, the harness raises the score from 0.650 to 0.704 (+5.4 points) and reduces tokens from 101.8K to 60.4K (a 41% reduction). Here, LLM calls increase from 4.5 to 6.7 per query, so the lower token count reflects fewer tokens per call, consistent

[1]We use the cleaned LongMemEval-S ground truth released at `https://huggingface.co/datasets/agentscope-ai/ReMe_longmemeval_clean_s_v2`.

Table 2: Transfer across benchmarks.

Training set	None		LongMemEval-S		BEAM-100K	
Test set	Score	Total Tok.	Score	Total Tok.	Score	Total Tok.
LongMemEval-S	0.888	43.9K	0.916	17.9K	0.908	26.6K
BEAM-1M	0.650	101.8K	0.639	39.0K	0.704	60.4K

Table 3: Time for one formal evaluation as acceleration mechanisms are added cumulatively. Each reduction is measured relative to the preceding row.

Setting	LongMemEval-S			BEAM		
	Cases (Queries)	Time	Reduction	Cases (Queries)	Time	Reduction
Full, serial	250 (250)	153.5 h	–	20 (400)	14.4 h	–
+ Parallel	250 (250)	5.5 h	96%	20 (400)	87 min	90%
+ Sampling	28 (28)	37 min	89%	10 (100)	38 min	56%
+ Reuse	28 (28)	2 min	94%	10 (100)	8 min	78%

with more selective and precise evidence being returned by search. The score gain on histories about $10\times$ longer than those used for optimization provides evidence that the harness transfers to a substantially larger memory scale. Although ReMe is already the strongest baseline on both benchmarks, these gains show that HoMe can discover and implement substantial improvements to a strong, human-designed memory harness. Given unobserved cache reuse, we compare query-token counts only between ReMe and HoMe, which share the same agentic answering workflow.

Table 2 shows transfer across benchmarks. The harness optimized on BEAM-100K improves the LongMemEval-S score from 0.888 to 0.908 while reducing average query-time tokens consumption from 43.9K to 26.6K, very close to harness optimized on LongMemEval-S. In the reverse direction, the LongMemEval-S-optimized harness greatly reduces BEAM-1M tokens by 62%, but its score falls from 0.650 to 0.639. This may stem from the higher initial score on LongMemEval-S, which leaves less room for quality gains and shifts optimization toward token savings at the expense of performance on harder BEAM cases. Nevertheless, it remains on the score–token Pareto frontier.

4.2.2 Optimization Efficiency

Strong harnesses emerge within a few iterations. Figure 2 shows the evaluation results of BEAM candidates on full training set, measured retrospectively after optimization. The second candidate already lies on the score–token Pareto frontier, and candidates two through four offer different trade-offs, reaching a score up to 0.73 or a query cost as low as 31.8K tokens. This optimization discover competitive harnesses within only a few iterations. The final two candidates do not extend the frontier, that additional iterations may not improve the harness. As edits accumulate, optimization may become more sensitive to random variation in sampled evaluations and overfit those cases.

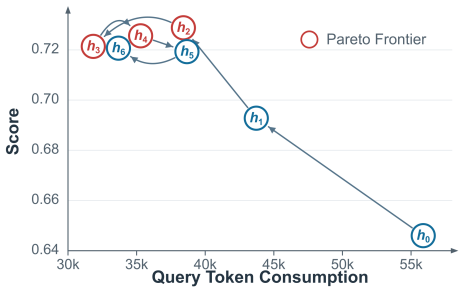

Figure 2: Scores and query-time token consumption of all candidates on BEAM. Scores are judge on full training set for each candidate. Upper left is better.

Sampling emphasizes informative failures. We compare the six BEAM candidates on their sampled evaluation sets and the complete training set. Query-token reductions are strongly correlated across the two settings (Pearson $r = 0.870$), so the smaller evaluations capture the direction of cost changes in this run. Answer scores are nearly uncorrelated ($r = -0.083$) but sampled evaluation scores are on average 0.05 lower than scores on the complete training set, consistent with the sampler's emphasis on historically difficult cases rather than a representative score estimate. The sampled cases provide diagnostic evidence rather than a direct ranking of candidates.

Formal evaluation time falls sharply. Table 3 shows the cumulative effects of parallelism, sampling, and construction reuse on one formal evaluation. Parallelism cuts full evaluation time by over

90% on both datasets; sampling processes fewer cases, and reuse avoids repeated construction. The 94% reduction from reuse on LongMemEval-S highlights the construction cost of its single-query cases. Reuse applies to five of six LongMemEval-S candidates and four of six BEAM candidates. Overall, the end-to-end optimization takes 13 hours 5 minutes on LongMemEval-S and 10 hours 26 minutes on BEAM. The Pareto frontier is reached after 6 hours 17 minutes on BEAM.

4.3 Case Study

HoMe improves the memory harness through changes to memory construction, retrieval code, and the system prompt. Below, we present four representative improvements. The first two improvements were independently discovered on different datasets, while the last two are harness optimizations specific to each datasets. The remaining improvements are documented in Appendix A.2.

1. **Improving cross-session evidence coverage.** The Diagnoser Agent found recurring errors on summarization and enumeration questions spanning multiple dates. Hybrid search returned relevant passages but, within its small retrieval window, often missed other pertinent sessions. Although memory construction had produced daily topic summaries, these competed with raw conversations in the same search and were not reliably retrieved. The Optimizer Agent changed the search tool to compile the daily summaries into a compact, date-grouped topic list. Each entry provides a description and source path. The answering prompt directs the Answer Agent to survey this list, then read the indicated files for details, rather than repeatedly searching for isolated passages.

2. **Diversifying the retrieval window.** The Diagnoser Agent observed that hybrid search often retrieved multiple chunks from the same file. Long raw transcripts were especially likely to occupy several retrieval slots, leaving the Answer Agent with fewer distinct sources. The Optimizer Agent introduced source-aware selection to encourage diversity. Each file receives a slot limit, stricter for raw transcripts than for memory notes. Once a file reaches its limit, chunks from other files fill the remaining slots. This prevents a single source from dominating retrieval.

3. **Indexing user actions on LongMemEval-S.** The Diagnoser Agent found that many questions require precise information about actions the user took, which can be difficult to locate within broader session notes. The Optimizer Agent therefore added a dedicated line to each note during memory construction that summarizes the user's own actions. It also built a separate inverted index over these lines, allowing the Answer Agent to search action descriptions and identify the notes containing potentially relevant evidence before reading them in full.

4. **Compressing retrieved evidence on BEAM.** To reduce the context returned by search, the Optimizer Agent introduced query-guided line selection within retrieved chunks. The search tool scores each line using keywords from the question and replaces lower-scoring lines with placeholders that preserve their line numbers. The Answer Agent thus receives a shorter search result while retaining access to the omitted text. If it needs more context, it can inspect the corresponding source file.

Together, these studies show that HoMe can use evaluation evidence to diagnose specific harness failures and translate those diagnoses into new code-level mechanisms and memory structures. Independent optimization runs on the two benchmarks arrived at related mechanisms, suggesting that source diversity and broader access to stored evidence may offer general design principles for memory systems. The benchmark-specific changes further show that HoMe can detect distinctive patterns in each dataset's questions and histories and adapt the harness to those patterns.

5 Conclusion

This paper presents HoMe, a novel evaluation-driven framework that optimizes the harness of a memory system without constraining the editable components or the search procedure. HoMe traces failures to the responsible sessions and harness components through session-level memory versioning and reduces evaluation cost of candidate harnesses through smoke tests, candidate gating, adaptive sampling, and memory reuse. With only six evaluated candidates per benchmark, HoMe extends the strong original harness of ReMe with new mechanisms such as fact-level memory indexes and source-aware retrieval. The resulting harnesses improve end-task accuracy on LongMemEval-S and BEAM by up to 5.4 percentage points while reducing query-time tokens by up to 59%. These results show that HoMe makes automated harness optimization practical even for strong memory systems.

AI USE STATEMENT

In this work, we used generative AI tools to assist with writing parts of the code and analyzing selected case studies. The authors conducted rigorous code reviews of all AI-assisted code and verified its correctness. We independently checked the AI-assisted case study analyses against the underlying evidence and wrote the corresponding discussion ourselves. Generative AI tools were also used for minor language polishing of the manuscript. All AI-assisted content was reviewed and verified by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The experimental setup and dataset splits are described in the main paper and the appendix. The code is available at `https://anonymous.4open.science/r/HoMe-EEC6/README.md`, where we will provide the code and detailed instructions needed to reproduce the reported results.

To help reproduce or improve upon the reported results, we recommend adapting ϵ and α to the difficulty of each benchmark. For easier benchmarks with high baseline scores, a smaller ϵ and a larger α can help avoid sampling too few failure cases. For harder benchmarks with lower baseline scores, a larger ϵ and a smaller α are recommended. An optional temperature parameter can also be introduced into the sampling distribution in Eq. 4. Increasing the temperature value can reduce repeated sampling of difficult cases for which multiple improvement attempts have failed.

REFERENCES

Zouying Cao, Jiaji Deng, Li Yu, Weikang Zhou, Zhaoyang Liu, Bolin Ding, and Hai Zhao. Remember me, refine me: A dynamic procedural memory framework for experience-driven agent evolution. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 16803–16822, San Diego, California, United States, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.829. URL https://aclanthology.org/2026.findings-acl.829/.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

DeepSeek-AI. Deepseek harness: Everything is a plugin. https://github.com/deepseek-ai/deepseek-harness, 2026.

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, 2025. URL https://arxiv.org/abs/2510.18866.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models, 2024. URL https://arxiv.org/abs/2405.14831.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, Senjie Jin, Jiejun Tan, Yanbin Yin, Jiongnan Liu, Zeyu Zhang, Zhongxiang Sun, Yutao Zhu, Hao Sun, Boci Peng, Zhenrong Cheng, Xuanbo Fan, Jiaxin Guo, Xinlei Yu, Zhenhong Zhou, Zewen Hu, Jiahao Huo, Junhao Wang, Yuwei Niu, Yu Wang, Zhenfei Yin, Xiaobin Hu, Yue Liao, Qiankun Li, Kun Wang, Wangchunshu Zhou, Yixin Liu, Dawei Cheng, Qi Zhang, Tao Gui, Shirui Pan, Yan Zhang, Philip Torr, Zhicheng Dou, Ji-Rong Wen, Xuanjing Huang, Yu-Gang Jiang, and Shuicheng Yan. Memory in the age of ai agents, 2025. URL https://arxiv.org/abs/2512.13564.

Sangyeop Kim, Yohan Lee, Sanghwa Kim, Hyunjong Kim, and Sungzoon Cho. Pre-storage reasoning for episodic memory: Shifting inference burden to memory for personalized dialogue. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 22096–22113. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.1204. URL http://dx.doi.org/10.18653/v1/2025.findings-emnlp.1204.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

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

Jiaqi Liu, Xinyu Ye, Peng Xia, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Evolvemem:self-evolving memory architecture via autoresearch for llm agents, 2026b. URL https://arxiv.org/abs/2605.13941.

Qingshan Liu, Guoqing Wang, Wen Wu, Jingqi Huang, Xinqi Tao, Dejia Song, Jie Zhou, and Liang He. Mempro: Agentic memory systems as evolvable programs, 2026c. URL https://arxiv.org/abs/2606.00619.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2023. URL https://arxiv.org/abs/2310.08560.

Wenbo Pan, Shujie Liu, Xiangyang Zhou, Shiwei Zhang, Wanlu Shi, Mirror Xu, and Xiaohua Jia. M*: Every task deserves its own memory harness, 2026. URL https://arxiv.org/abs/2604.11811.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Xufang Luo, Hao Cheng, Dongsheng Li, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Jianfeng Gao. On memory construction and retrieval for personalized conversational agents, 2025. URL https://arxiv.org/abs/2502.05589.

Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory, 2025. URL https://arxiv.org/abs/2501.13956.

Jingwei Sun, Jianing Zhu, Jiangchao Yao, Tongliang Liu, and Bo Han. Rethinking how to remember: Beyond atomic facts in lifelong llm agent memory, 2026. URL https://arxiv.org/abs/2605.19952.

Mohammad Tavakoli, Alireza Salemi, Carrie Ye, Mohamed Abdalla, Hamed Zamani, and J Ross Mitchell. Beyond a million tokens: Benchmarking and enhancing long-term memory in llms, 2025. URL https://arxiv.org/abs/2510.27246.

Yu Wang and Xi Chen. Mirix: Multi-agent memory system for llm-based agents, 2025. URL https://arxiv.org/abs/2507.07957.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory, 2024. URL https://arxiv.org/abs/2410.10813.

Di Wu, Zixiang Ji, Asmi Kawatkar, Bryan Kwan, Jia-Chen Gu, Nanyun Peng, and Kai-Wei Chang. Longmemeval-v2: Evaluating long-term agent memory toward experienced colleagues, 2026. URL https://arxiv.org/abs/2605.12493.

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

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, Xiangzheng Zhang, and Tong Yang. Harness-bench: Measuring harness effects across models in realistic agent workflows, 2026. URL https://arxiv.org/abs/2605.27922.

Yunbei Zhang, Janet Wang, Yingqiang Ge, Weijie Xu, Jihun Hamm, and Chandan K. Reddy. Stop comparing llm agents without disclosing the harness, 2026. URL https://arxiv.org/abs/2605.23950.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents, 2024. URL https://arxiv.org/abs/2404.13501.

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, 2023. URL https://arxiv.org/abs/2306.05685.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory, 2023. URL https://arxiv.org/abs/2305.10250.

Sizhe Zhou, Sheldon Yu, Hui Wei, Junda Wu, Siru Ouyang, Yizhu Jiao, Shijia Pan, Julian McAuley, Yu Zhang, Tong Yu, and Jiawei Han. Filesystem-based memory for llm agents: Organization, evolution, and sustainability, 2026. URL https://arxiv.org/abs/2607.26637.

A Appendix

A.1 Per-Category Results

Tables 4 and 5 report answer scores by question type.[2] Relative to ReMe, HoMe improves LongMemEval-S most on single-session preference questions (score 0.667 to 0.933; tokens 31.4K to 11.8K), with smaller score gains on knowledge update and temporal reasoning. Token use falls in all six categories, including those with unchanged scores, but the multi-session score slips from 0.836 to 0.806 as tokens fall from 54.5K to 19.3K. On BEAM-1M, the largest gains occur in contradiction resolution (0.391 to 0.705; 50.6K to 22.4K) and abstention (0.429 to 0.636; 119.9K to 43.4K). Conversely, knowledge update and information extraction lose 0.108 and 0.074 score points, respectively, despite substantial token reductions; these cases expose a possible cost of more selective retrieval or shorter query traces. Across both benchmarks, nine of the ten categories with higher scores also use fewer query-stage tokens, so the improvements are generally accompanied by lower consumption rather than larger token budgets. BEAM summarization is the exception: its score rises from 0.693 to 0.726 while tokens rise from 161.7K to 173.2K. Because these totals aggregate multiple calls, they cannot by themselves establish whether each call saw less context, or which harness change caused a gain or regression.

A.2 Case Studies of Memory System Optimization

This appendix summarizes memory harness improvements from the LongMemEval-S and BEAM optimization runs, focusing on the failures they address and the mechanisms behind them.

A.2.1 LongMemEval-S: Fact Recall, Aggregate Retrieval, and Answer Calibration

1. **Write short, self-contained user facts instead of topic-oriented essays.** Baseline notes often devoted space to generic advice while losing exact dates, numbers, or the option the user actually chose. The surrounding generic text also diluted the embedding representation of specific facts. The revised construction policy prioritized complete user states, preferences, decisions, and feelings; preserved session-specific names, dates, and prices verbatim; and compressed generic explanations to a line. It aimed for one note per session, beginning with self-contained key user facts. This reduced overlapping notes and made important facts more likely to survive chunking and be retrieved on their own. When a length budget is tight, generic background should be removed before user facts.

[2] Category abbreviations: Abs. = Abstention; CR = Contradiction Resolution; EO = Event Ordering; IE = Information Extraction; IF = Instruction Following; KU = Knowledge Update; MS = Multi-session; MSR = Multi-session Reasoning; PF = Preference Following; SSA = Single-session Assistant; SSP = Single-session Preference; SSU = Single-session User; Sum. = Summarization; TR = Temporal Reasoning.

Table 4: Per-category scores on LongMemEval-S.

Method	Metric	KU	MS	SSA	SSP	SSU	TR
A-MEM	Score	0.790	0.657	0.963	0.600	0.914	0.647
	Total Tok.	21.5K	21.5K	15.6K	18.6K	19.3K	19.4K
	In Tok.	21.5K	21.5K	15.5K	18.6K	19.2K	19.4K
	Out Tok.	< 0.1K	< 0.1K	< 0.1K	< 0.1K	< 0.1K	< 0.1K
	LLM Calls	2.0	2.0	2.0	2.0	2.0	2.0
SimpleMem	Score	0.711	0.761	0.926	0.533	0.800	0.632
	Total Tok.	35.7K	37.5K	28.9K	47.7K	37.2K	40.5K
	In Tok.	27.0K	28.3K	21.2K	36.1K	29.1K	31.1K
	Out Tok.	8.7K	9.2K	7.6K	11.6K	8.1K	9.4K
	LLM Calls	7.8	8.0	7.0	8.9	7.9	8.6
ReMe	Score	0.921	0.836	1.000	0.667	0.971	0.882
	Total Tok.	29.0K	54.5K	15.8K	31.4K	26.6K	65.7K
	In Tok.	28.5K	53.2K	15.5K	30.6K	26.3K	64.8K
	Out Tok.	0.6K	1.3K	0.3K	0.7K	0.3K	0.9K
	LLM Calls	3.8	5.2	2.9	4.1	3.5	6.2
HoMe	Score	0.974	0.806	1.000	0.933	0.971	0.940
	Total Tok.	9.7K	19.3K	9.8K	11.8K	11.2K	29.2K
	In Tok.	9.3K	18.3K	9.5K	11.0K	11.0K	28.5K
	Out Tok.	0.5K	1.0K	0.3K	0.9K	0.2K	0.7K
	LLM Calls	2.2	3.3	2.2	2.7	2.3	4.0

Table 5: Per-category scores on BEAM-1M.

Method	Metric	Abs.	CR	EO	IE	IF	KU	MSR	PF	Sum.	TR
A-MEM	Score	0.629	0.505	0.486	0.774	0.689	0.779	0.661	0.840	0.540	0.518
	Total Tok.	30.9K	28.4K	27.4K	27.6K	26.7K	29.7K	33.2K	31.3K	35.2K	32.7K
	In Tok.	29.7K	27.2K	24.1K	26.3K	24.7K	28.8K	31.1K	29.2K	31.4K	31.6K
	Out Tok.	1.2K	1.2K	3.3K	1.3K	2.0K	0.8K	2.1K	2.1K	3.8K	1.1K
	LLM Calls	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
SimpleMem	Score	0.564	0.241	0.393	0.544	0.349	0.546	0.356	0.624	0.459	0.324
	Total Tok.	51.7K	24.4K	31.3K	28.9K	45.6K	37.1K	39.7K	34.9K	28.0K	26.4K
	In Tok.	40.7K	16.7K	20.2K	20.4K	35.0K	27.9K	28.9K	25.6K	18.2K	18.4K
	Out Tok.	11.0K	7.7K	11.1K	8.5K	10.5K	9.2K	10.8K	9.3K	9.8K	8.0K
	LLM Calls	9.6	5.9	6.2	6.4	7.6	7.5	6.9	6.4	5.6	6.0
ReMe	Score	0.429	0.391	0.558	0.809	0.852	0.779	0.658	0.798	0.693	0.536
	Total Tok.	119.9K	50.6K	205.4K	79.8K	56.7K	46.6K	141.0K	52.7K	161.7K	103.3K
	In Tok.	118.7K	49.8K	201.5K	78.9K	55.8K	46.0K	138.1K	51.8K	158.8K	100.2K
	Out Tok.	1.2K	0.8K	3.9K	0.9K	0.9K	0.7K	2.9K	0.9K	2.9K	3.1K
	LLM Calls	5.2	3.5	5.8	4.0	3.8	3.4	5.4	3.5	5.4	4.9
HoMe	Score	0.636	0.705	0.546	0.735	0.901	0.671	0.682	0.833	0.726	0.604
	Total Tok.	43.4K	22.4K	123.0K	32.5K	30.7K	19.2K	93.5K	25.0K	173.2K	41.4K
	In Tok.	41.9K	21.0K	116.6K	31.2K	29.4K	18.3K	88.1K	23.8K	169.0K	39.1K
	Out Tok.	1.5K	1.4K	6.4K	1.2K	1.3K	0.9K	5.4K	1.2K	4.2K	2.3K
	LLM Calls	6.6	4.6	12.7	5.2	5.5	4.0	8.5	4.8	9.4	6.0

2. **Control both semantic duplication and concentration by source in a small window.** Similar notes from one session can consume nearly every retrieval slot. Multiple chunks from one transcript can also occupy several slots before rendering merges them into a single displayed item, leaving no room for a different source. The first problem was addressed with relevance-heavy maximal marginal relevance: highly similar candidates receive a modest penalty while strong retrieval scores still dominate. The second was addressed by capping the slots taken by each file, with a stricter cap for raw transcripts than for notes. Both mechanisms postpone excess candidates rather than deleting them. The useful accounting unit is the *distinct evidence ultimately delivered*, not merely the number of retrieved chunks.

3. **Add an inexpensive catalog index beyond top-k retrieval.** A question such as "How many times did I attend a graduation?" requires every occurrence, whereas a similarity search guarantees only the most similar few. Each note already has a title and a one-line description, so the system can scan this compact catalog across all notes, match query terms, and append a candidate

list to normal search results. A document-frequency threshold removes generic template words such as "user"; ranking uses the strongest discriminative term rather than allowing many weak matches to accumulate. Each entry keeps its path and a description excerpt around the match, telling the agent which note to inspect. The index addresses a *coverage gap* in top-k retrieval while leaving semantic search in place.

4. **Extend the catalog to individual facts while retaining both levels.** A one-line description is necessarily lossy: an additional graduation, a particular island, or a precise bedtime may appear only in the note body. The expanded index scans notes and splits self-contained Markdown list items into fact entries. Matching entries are quoted inline, allowing the agent to use the fact without another read call. Fact and catalog layers calculate term frequencies and rankings separately, so adding the finer layer does not silently change how the coarser one behaves. Facts already visible in the normal retrieval window are suppressed in the index to avoid duplication. The catalog remains useful because a question phrased in terms of a session's topic may share no content word with the answer sentence. The layers answer complementary questions: "Which note might matter?" and "Which statement in it matters?"
5. **Keep the inexpensive index available, vary its width, and state what a match proves.** The run briefly gated index delivery to enumeration questions or explicit requests to reduce irrelevant matches. Case-by-case inspection showed that the gate did not cause the improvements initially attributed to it; withholding the index instead prompted repeated blind query rewrites and higher total token use. The final design kept the index available for every question, returned a shorter list for point queries and a wider one for enumeration, and warned in the index header that keyword overlap is only a lead. A nonempty list does not establish that the entity or event in the question exists; each entry must be checked against the requested fact. Query wording alone did not predict where evidence was stored. For a cheap index, controlling output size and clarifying evidential meaning proved more useful than gating access by question form.
6. **Diagnose "the evidence was found" separately from "the answer was correct."** In several cases the agent read the right note but abstained because it demanded a completion confirmation that memory would never record. In others, it corrected a false premise yet still supplied the value that the false premise requested. Answering rules were calibrated to anchor relative times, use newer records for current state, respect entity boundaries, and reject the requested value when its premise is false. Rules without case-level evidence were later removed because their prompt cost recurred at every ReAct step. A memory answer can fail at construction, retrieval, evidence interpretation, or final decision. More retrieved material is unlikely to repair a decision error once the correct evidence is already visible.

A.2.2 BEAM: Coverage, Order, and Access to Evidence in Long Conversations

1. **Reserve room for both curated memory and primary evidence in the retrieval window.** The system stores condensed notes alongside verbatim transcripts, which account for roughly 95% of indexed chunks. A window filled solely by the highest-scoring chunks is therefore easily dominated by transcripts, leaving out the notes that integrate information across passages. The optimized ranking favors notes while reserving slots for transcripts, so the answering agent can use the consolidated account and still check exact wording, numbers, and chronology. This rearrangement changes the order of candidates without discarding any. Crucially, it happens before the system marks returned chunks as seen: otherwise, a chunk displaced from the window could be recorded as already shown and become unreachable in later searches. In a mixed memory store, the *source composition of the window* matters as much as the ranking of individual chunks.
2. **Compress search responses without cutting off access to the original text.** Long transcript chunks make repeated searches expensive, yet a simple truncation can hide the line containing the answer. The first measure placed a character budget on responses while retaining the path and line range for each result; text that did not fit remained available through an explicit pointer for targeted reading. When truncation from the beginning repeatedly omitted facts in the middle or end of a note, selection moved to the line level. Lines were scored using the distinctiveness of query terms, numbers, and heading cues. The budget first covered multiple evidence-bearing lines, then used any remaining space for adjacent context. When a semantic match had no lexical overlap, the title and summary were still retained. Effective compression thus has two requirements: show the most useful evidence now, and make every omitted passage straightforward to locate later.

Table 6: LongMemEval-S optimization trajectory.

Cand.	Elapsed	Lines (+/–)	Changed	Score	Main change
L1	6:15	$+313/-15$	$cp\theta$	0.607	Rebalanced retrieval by relevance and source; revised memory construction and answering rules.
L2	9:27	$+318/0$	$c\theta$	0.630	Added a compact index of memory-note descriptions.
L3	10:41	$+321/-118$	$c\theta$	0.593	Extended the index to individual facts.
L4	12:03	$+80/-53$	$cp\theta$	0.593	Gated index delivery and adjusted evidence interpretation rules.
L5	12:35	$+19/-22$	p	0.630	Reverted broader prompt changes while retaining targeted fixes.
L6*	13:05	$+73/-47$	$c\theta$	0.741	Restored the always-available index with query-dependent width.

3. **Give coverage questions a global view instead of repeatedly enlarging top-k.** Answers to questions such as "Which topics did I mention?" may be spread across many sessions. Similarity search can return highly relevant chunks without establishing that all matching occurrences have been found. The optimizer therefore derived a compact chronological index from *all* notes, appended it to ordinary retrieval results, and delivered it automatically at most once per question. Entries carried paths for verification. Near-duplicate entries from overlapping notes in one session were consolidated while their paths remained accessible. To control cost, automatic delivery was routed only for coverage questions after offline checks. A measured regression exposed the index's limit: a thematic summary can look sufficient for a question about the *first-mention order* while omitting precisely the ordering evidence needed to answer it. A global index can support "what was mentioned" without automatically serving as an answer to "in what order."
4. **For ordering questions, provide a reading route rather than a summary that invites an immediate answer.** Ordering evidence spans dates and sessions, so one note seldom contains all relevant mentions. An earlier summary-style index encouraged the agent to answer from its overview, saving tokens but getting the sequence wrong. The revised route listed note paths in chronological order and then annotated each path with the line range of its conversation-flow section. The agent could read the passages that recorded mention order and speaker identity without loading every entire note. Those sections occupied about 30% of a note on average, making broader coverage feasible within the tool and context budgets. An index need only guide reading; when the answer depends on fine-grained order, compressing evidence into a topical summary can remove the deciding information.
5. **Record relationships at construction time that retrieval cannot reconstruct.** Case-level traces showed failures in which the relevant note had been found, but the note never preserved *who* made a statement, *when* a topic first appeared, or whether a new statement superseded an older one. Construction rules were revised to identify the speaker, distinguish an assistant's suggestion from a user's decision, retain a conversation-flow record in actual mention order, preserve enduring instructions verbatim, and keep both sides of an update with an explicit supersession relation. Retrieval can reorder stored text; it cannot recover attribution, order, or update relations discarded during compression. The questions a memory system must answer should therefore inform the structure of what it writes.

Together, these runs show that memory optimization reaches beyond vector ranking. Construction determines which relationships remain recoverable; the retrieval window determines which sources fit within a budget; derived indexes address the coverage limits of top-k search; and answer policy determines how evidence is used. When validation rounds sample different cases, contributions of individual mechanisms are best assessed through paired questions and execution traces, particularly when several changes were introduced together.

A.3 Optimization Trajectories

Tables 6 and 7 summarize the candidates proposed in each optimization run. Scores are measured on sampled training questions; $\star$ marks the final harness selected by the Optimizer Agent. Elapsed is the cumulative wall-clock time at the completion of each iteration, including diagnosis, candidate

Table 7: BEAM optimization trajectory.

Cand.	Elapsed	Lines (+/–)	Changed	Score	Main change
B1	5:00	$+165/-1$	$cp\theta$	0.694	Rebalanced notes and transcripts; revised construction and answering rules.
B2	6:17	$+216/-19$	$cp\theta$	0.698	Selected evidence lines using the query.
B3	6:54	$+377/-1$	$cp\theta$	0.710	Added a cross-session topic index.
B4*	8:10	$+94/-18$	cp	0.670	Narrowed index routing and consolidated duplicate notes.
B5	9:16	$+140/-22$	cp	0.715	Replaced the topic overview with a chronological source-path map.
B6	10:10	$+66/-23$	cp	0.677	Added conversation-flow line ranges to the map.

generation, evaluation, and the initial validation reused by these runs. The initial validation took 5 hours 30 minutes on LongMemEval-S and 87 minutes on BEAM. Line counts compare each candidate with its predecessor. In the Changed column, c, p, and θ denote changes to code, prompts, and configuration, respectively.

A.4 Adaptive Sampling Details

The initial full training-set evaluation initializes a sampling record for each case and query. For a sampling unit v, the record contains its exponential moving average of loss $\bar{\ell}_v$, number of valid score observations n_v, most recent selected round t_v^{sel}, most recent error round t_v^{err}, and number k_v of perfect-score observations since that error. Only valid scores update $\bar{\ell}_v$ and n_v. A case-level observation is the mean score of the queries evaluated for that case in the current round.

After observing score $y_v^{(b)}$ in round b, the sampler updates

$$\bar{\ell}_v \leftarrow \begin{cases} 1 - y_v^{(b)}, & n_v = 0, \\ \alpha(1 - y_v^{(b)}) + (1-\alpha)\bar{\ell}_v, & n_v > 0, \end{cases} \qquad n_v \leftarrow n_v + 1. \tag{6}$$

A score below 1 sets $t_v^{\mathrm{err}} = b$ and $k_v = 0$; a perfect score after an earlier error increments k_v. Units without a valid observation retain their loss estimate and observation count.

For a unit eligible in round b, let $\Delta_v = \max(b - t_v^{\mathrm{sel}}, 0)$. The unresolved-error term in the sampling priority is

$$u_v = \begin{cases} 0, & \text{if } v \text{ has no recorded error,} \\ \dfrac{\log(1 + \max(b - t_v^{\mathrm{err}}, 0))}{1 + k_v}, & \text{otherwise.} \end{cases} \tag{7}$$

The priority weights are fixed at $\lambda_s = 0.2$, $\lambda_n = 0.2$, and $\lambda_u = 0.1$. For the eligible set $\mathcal{V}$, the sampling probability is

$$p_v = (1-\epsilon)\frac{\exp(r_v)}{\sum_{v' \in \mathcal{V}} \exp(r_{v'})} + \frac{\epsilon}{|\mathcal{V}|}, \tag{8}$$

where r_v is the priority defined in the main text.

The moving-average loss favors currently difficult units, while the staleness term revisits units that have not been tested against recent harness changes. The observation-count term gives less frequently measured units additional opportunity to be sampled. The unresolved-error term retains attention on an earlier failure until subsequent perfect-score observations reduce its contribution. Logarithmic staleness growth prevents age alone from increasing priority linearly, and the uniform component gives every eligible unit a nonzero chance of exposing a new regression. These terms are sampling heuristics rather than estimates of statistical uncertainty.

Sampling proceeds at two levels. The sampler first sets $\mathcal{V} = \mathcal{D}$ and draws cases without replacement according to p_v. If the queries in the selected cases exceed the per-iteration query cap, it sets $\mathcal{V}$ to the queries in those cases and applies the same sampling rule at the query level. Otherwise, it evaluates all their queries. Both levels deliberately favor historically difficult units. Consequently, the resulting score is useful for finding failures and guiding revisions, but should not be interpreted as an unbiased estimate of a candidate's full-training-set performance.