AdaMemPIP: Adaptive Memory Pipeline Orchestration for LLM Agents

Anonymous ACL submission

Abstract

Large language model (LLM) agents increasingly depend on external memory to support long-horizon interaction. Yet most current memory systems still use fixed memory rules. They process each new dialogue with the same extractor, retrieve each query with the same method, and present evidence in the same format. This is restrictive. Different memory needs require different choices. We propose AdaMemPIP, an adaptive memory orchestration framework for LLM agents. AdaMemPIP uses an LLM to route both memory storage and memory usage. For storage, it selects extractors for each new dialogue record and writes the extracted items to matched memory libraries. For usage, it selects retrieval methods, target libraries, and organization strategies for each query, then executes the resulting pipeline. The framework is also extensible: new extractors, retrievers, stores, and organizers can be added without changing the overall control logic. This makes AdaMemPIP a flexible and scalable approach to long-term agent memory.

1 Introduction

Large language model (LLM) agents need memory to work over long interaction horizons (Park et al., 2023; Packer et al., 2023; Zhong et al., 2024; Xu et al., 2025; Chhikara et al., 2025). They must remember user facts, update outdated information, recover earlier assistant advice, and connect evidence across sessions. In short, they must act over time, not only over one prompt.

Most current systems still use a largely fixed process. They extract each dialogue in one default way, store it in one default structure, and retrieve each query in one default way (Packer et al., 2023; Zhong et al., 2024; Chhikara et al., 2025; Fang et al., 2025). This is easy to implement, but too rigid. A temporal question, a preference question, and a question about what the assistant said earlier do not need the same memory behavior. One fixed routine can waste computation in some cases and lose useful structure in others.

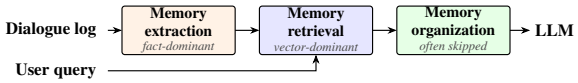

Figure 1: A simplified view of common memory-management pipelines. Many systems use a fixed sequence of extraction, retrieval, and reorganization, with a dominant default choice at each step.

This matters for both quality and cost. Recent benchmarks show that long-term interactive memory remains hard even for strong models (Wu et al., 2025; Maharana et al., 2024; Tan et al., 2025). At the same time, always running a rich memory pipeline is expensive (Fang et al., 2025). The problem is therefore not only how to build better memory modules. It is also how to decide what to store and how to use it later.

We treat memory management as a *control problem*. The key question is not which single store or retriever is best. The key question is how an agent should route storage and usage decisions under changing needs. For a new dialogue record, it should decide which extractor to run and where to write the result. For a new query, it should decide which memory libraries to search, whether to use vector or keyword retrieval, and how to organize the evidence. Recent work on adaptive memory also points to the value of selectivity, but usually focuses on one mechanism or one update policy (Wu et al., 2026; Ma et al., 2026; Su et al., 2026). Figure 1 shows the fixed-process pattern that remains common in existing systems.

To address this limitation, we propose AdaMemPIP, an adaptive memory pipeline for LLM agents. AdaMemPIP uses an LLM controller to route modular memory operators. It has two paths. In the storage path, the controller selects extractors and writes their outputs to matched memory libraries.

In the usage path, it chooses memory libraries, retrieval methods, and a reader. In our implementation, the reader is one of three lightweight options: flat, assistant-focused, or timeline.

This design has two benefits. It matches the memory path to the current need, and it stays extensible. Each extractor, retriever, store, or reader is a callable function in a library. New operators can be added without redesigning the control logic.

We evaluate AdaMemPIP on LongMemEval (Wu et al., 2025) and LoCoMo (Maharana et al., 2024). AdaMemPIP achieves the best total score on LongMemEval for both tested backbones, shows encouraging transfer to LoCoMo, and remains much cheaper than heavy structured-memory systems. These results support a simple claim: for long-term agents, the hard part is not only storing more memory. It is choosing the right memory path for the current turn. A useful agent must therefore manage memory conditionally, not uniformly.

Our contributions are:

- We identify adaptive memory orchestration as a core problem for long-term LLM agents, and we show why fixed memory rules for both storage and usage are fundamentally limited.
- We propose AdaMemPIP, a modular framework with two LLM-routed paths: adaptive memory storage and adaptive memory usage. The design naturally supports future extension with new memory-management methods.
- We show across multiple benchmarks, ablations, and efficiency analyses that adaptive orchestration improves memory quality while keeping memory cost practical.

2 Related Work

2.1 Memory Architectures for LLM Agents

Long-term memory is now a core part of LLM agents. Prior work studies explicit memories (Park et al., 2023; Zhong et al., 2024), hierarchical access (Packer et al., 2023), agentic memory systems (Xu et al., 2025), scalable production memory (Chhikara et al., 2025), and more structured or personalized memory forms (Wang et al., 2024; Zhang et al., 2024; Westhäußer et al., 2025; Lu and Li, 2025). These studies show that memory representation matters. However, most of them still use a mostly fixed control flow after the modules are defined. Our work instead focuses on *adaptive orchestration* across memory operations.

2.2 Efficient and Adaptive Memory Use

Another line of work studies efficient and selective memory use. LightMem focuses on lightweight memory construction (Fang et al., 2025). Other work studies memory refinement, strategy-guided recall, and more adaptive memory control (Salama et al., 2025; Zhao et al., 2025; Wu et al., 2026; Ma et al., 2026; Su et al., 2026). These methods highlight the value of selectivity, but they usually optimize one mechanism or one update policy. In contrast, AdaMemPIP asks a broader question: given several candidate extractors, retrievers, stores, and readers, which ones should be activated for the current request?

2.3 Memory Benchmarks and Analysis

Progress in agent memory has also been driven by stronger benchmarks. LongMemEval and LoCoMo test long-term interactive memory under different question types and conversation lengths (Wu et al., 2025; Maharana et al., 2024). MemBench and mtRAG broaden this view (Tan et al., 2025; Katsis et al., 2025). Other work studies memory quality more directly, including the relation between contextual and parametric memory and the integrity of persistent memory (Tao et al., 2024; Mireshghallah et al., 2025). Together, these benchmarks support the view that agent memory is not one skill. Different questions stress different memory operations.

3 Method

AdaMemPIP separates memory *operators* from memory *control*. The operators extract, store, retrieve, and read memory, as shown in Figure 2. The controller decides which operators to call for the current dialogue record or query. This keeps the framework modular, makes adaptivity explicit, and lets us add new memory operators without changing the overall control logic.

3.1 Problem Setup

We consider an LLM agent that interacts with a user over multiple sessions. At turn t, the agent observes a new dialogue record D_t, a current user query q_t, and the memory state M_t. The goal is to answer q_t and update memory for future turns. AdaMemPIP therefore solves two routing problems at each turn: how to extract and store memory from D_t, and how to retrieve and read memory for q_t. Let $\mathcal{F}^{\text{ext}}$, $\mathcal{F}^{\text{ret}}$, $\mathcal{F}^{\text{store}}$, and $\mathcal{F}^{\text{org}}$ denote the candidate function sets for extraction, retrieval, storage, and evidence

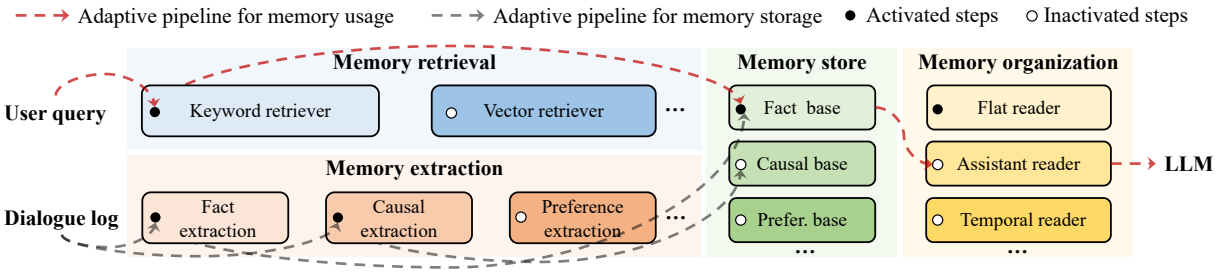

Figure 2: Overview of AdaMemPIP. The system uses an LLM to route both memory storage and memory usage. It adaptively selects extraction for new dialogue records, and retrieval plus evidence reading for user queries.

organization. A memory step is a function

$$f : \mathcal{X} \to \mathcal{X},$$

where the state space $\mathcal{X}$ contains dialogue records, queries, memory state, and intermediate evidence. The core problem is not to fix one operator per stage in advance. The core problem is to let an LLM choose a small task-matched set of functions online. In our implementation, the memory state is partitioned into typed libraries,

$$M_t = \{M_t^{\text{fact}}, M_t^{\text{pref}}, M_t^{\text{causal}}\},$$

because different questions need different memory views. Unlike a fixed pipeline, AdaMemPIP routes memory operations according to the current need.

We make this formulation concrete in two ways. First, each extractor maps a dialogue record to a set of typed memory items,

$$f^{\text{ext}} : D_t \to E_t, \qquad E_t = \{e_1, \ldots, e_n\},$$

where each item stores its type, content, and source record. The source record is kept so that later modules can recover timestamps or assistant-side text when needed. Second, a usage plan is a short list of retrieval calls,

$$\mathbf{r}_t = \left[(\ell_i, m_i, k_i)\right]_{i=1}^{n_t},$$

where ℓ_i is a memory library, $m_i \in \{\text{vector}, \text{keyword}\}$ is a retrieval method, and k_i is an optional keyword phrase. The controller then chooses one reader to organize the aggregated evidence for generation.

3.2 Overview

Figure 2 shows the full framework. AdaMemPIP contains four libraries: memory extraction, memory retrieval, memory store, and memory organization. The framework runs two coupled but asymmetric processes. The storage path starts from D_t. The controller selects one or more extractors, and the extracted memory items are then written to matched stores. The usage path starts from q_t. The controller first selects retrieval methods and memory libraries, then picks one reader. The retrieved evidence is organized by that reader and passed to the backbone LLM. The two paths share the same memory state, but solve different decisions. Storage asks what should be written into memory. Usage asks what should be retrieved and how the evidence should be read before generation.

3.3 Function Libraries

We define four libraries of memory functions.

The extraction library is

$$\mathcal{F}^{\text{ext}} = \{f_{\text{fact}}, f_{\text{causal}}, f_{\text{pref}}\}.$$

These functions map dialogue context to reusable memory units. Fact extraction captures explicit user information and world knowledge. Causal extraction captures reasons, dependencies, and event consequences. Preference extraction captures stable user-specific tendencies.

The retrieval library is

$$\mathcal{F}^{\text{ret}} = \{f_{\text{kw}}, f_{\text{vec}}\}.$$

Keyword retrieval helps with explicit entity matching and precise lexical cues. Vector retrieval is better for semantic similarity and paraphrased recall. During memory usage, the controller decides which memory libraries the retrievers should query.

The storage library is

$$\mathcal{F}^{\text{store}} = \{f_{\text{fact-base}}, f_{\text{causal-base}}, f_{\text{pref-base}}, \ldots\}.$$

These stores hold different memory types. In our implementation, the target store is determined by the extractor output. Fact memories go to the fact base, causal memories go to the causal base, and preference memories go to the preference base.

The organization library is

$$\mathcal{F}^{\text{org}} = \{f_{\text{flat}}, f_{\text{assistant}}, f_{\text{timeline}}\}.$$

We implement these functions as three readers. *FlatReader* merges evidence into a flat passage. *AssistantReader* emphasizes assistant-side utterances when the query asks what the assistant said before. *TimelineReader* orders evidence by inferred event time and falls back to the original memory timestamp when event time cannot be inferred.

These libraries matter because long-term memory is heterogeneous. Facts, preferences, causal relations, lexical matches, semantic matches, and temporal structure do not serve the same role. We keep them separate so that the controller can choose the right behavior for the current need.

3.4 Adaptive Routing

All routing decisions are made by an LLM controller. We use prompt-based routing in both storage and usage. The controller outputs structured JSON plans that are executed directly. No task-specific fine-tuning is required. For memory storage, the controller reads D_t and M_t. It selects one or more extractors:

$$\mathbf{z}_t^{\text{store}} = g_{\text{store}}(D_t, M_t).$$

The activated extractors run independently on the dialogue record. Their outputs are then written to the matched libraries. Co-activation is allowed because one dialogue record may contain several reusable memory types, such as facts plus preferences or facts plus causal relations. For memory usage, the controller factorizes routing into two lightweight steps. First, a retrieval planner reads q_t and outputs one or more query objects:

$$\mathbf{r}_t = g_{\text{ret}}(q_t).$$

Each query object specifies a memory library, a retrieval method, and optional keyword phrases. The planner may emit multiple query objects, which allows the system to combine broad semantic recall with precise lexical matching. Keyword search is restricted to explicit noun or entity phrases, and we allow at most two keyword query objects in one plan. The current routing prompt also requires fact memory to be included at minimum, which makes fact memory the default anchor for retrieval. Then, a selector chooses one evidence reader:

$$z_t^{\text{read}} = g_{\text{read}}(q_t).$$

We use single-reader execution in all experiments, which yields a lightweight *Retrieve* $\rightarrow$ *Read* $\rightarrow$ *Generate* pipeline. FlatReader is the default fallback. AssistantReader is reserved for questions whose answer should come from a prior assistant utterance. TimelineReader is used for temporal questions that require event ordering. In our current implementation, the reader selector depends only on q_t. This design is intentional: question form is the main signal for deciding whether the evidence should be read as flat context, assistant-side text, or a timeline, while memory-dependent decisions are already handled by retrieval planning. The controller does not replace the memory operators. It decides which ones to call. This separation is the core design of AdaMemPIP. It also makes the framework easy to extend. New extractors, retrievers, stores, or readers can be added without changing the control logic. Appendix A shows the retrieval-routing prompt used in our experiments.

3.5 End-to-End Execution

Let E_t denote the union of all extracted memory items from the selected extractors. The storage path can then be written as

$$E_t = \bigcup_{f \in \mathbf{z}_t^{\text{store}}} f(D_t), \quad M_{t+1} = f^{\text{store}}(E_t, M_t),$$

where the selected extractors produce memory items and each item is written to its matched memory library. Let C_t denote the aggregated evidence returned by all retrieval calls in $\mathbf{r}_t$. The usage path can be written as

$$\begin{aligned} C_t &= \text{Agg}\left(\{f_{m_i}^{\text{ret}}(q_t, M_t^{\ell_i}; k_i)\}_{i=1}^{n_t}\right), \\ \hat{e}_t &= f_{z_t^{\text{read}}}^{\text{org}}(C_t), \end{aligned}$$

where $\text{Agg}(\cdot)$ aggregates the outputs of all selected retrieval calls, and $\hat{e}_t$ is the organized evidence passed to the backbone LLM. The final answer is then generated from the current query, the dialogue context, and $\hat{e}_t$. This is the main difference from prior systems. A fixed pipeline commits to one default extractor, retriever, and reader. AdaMemPIP does not. The runtime behavior is simple. A new dialogue record triggers the storage path. A user query triggers the usage path. In both cases, the controller makes the routing decision first. The selected modules are then executed. The backbone LLM finally receives the current dialogue context and the selected memory evidence, and produces

the response. In summary, AdaMemPIP is an LLM-routed system with one adaptive path for storage and another for usage.

3.6 Efficiency Discussion

AdaMemPIP adds a control step before memory operations, so routing is not free. During storage, the controller reads the new dialogue record and selects extractors. During usage, it reads the query and produces a short retrieval plan plus one reader choice. We keep this overhead bounded by using compact JSON plans, a small operator set, and single-reader execution. The goal is to trade this small control cost for lower downstream memory cost. A fixed rich pipeline may extract every memory type, search every memory library, and process evidence in the same way for every query. AdaMemPIP can skip irrelevant extractors, avoid unlikely memory libraries, and use FlatReader when no temporal or assistant-side reading is needed. This benefit is instance-dependent: hard cases may still activate several modules. The expected gain comes from mixed workloads, where many turns need only a narrow memory path. Figure 5 and the activation analysis examine this tradeoff empirically.

4 Experiments

We seek to answer the following research questions through our experiments:

- Does adaptive orchestration improve long-term memory quality over fixed pipelines?
- Do the gains transfer across backbone models and benchmarks?
- Which adaptive components matter most?
- Does the method remain efficient despite the extra routing step?

4.1 Datasets

We evaluate on two benchmarks for long-term conversational memory: LongMemEval (Wu et al., 2025) and LoCoMo (Maharana et al., 2024). We keep the official split and evaluation protocol for both datasets unchanged. LongMemEval is our main benchmark. It evaluates long-term interactive memory for chat assistants and separates evaluation into six question types: Temporal, Multi-Session, Knowledge-Update, Single-User, Single-Assistant, and Single-Preference. This categorization is important for our study because these question types stress different memory behaviors. LoCoMo serves as an additional cross-benchmark test of generalization. It focuses on very long-term conversational memory and requires models to answer questions from extended multi-session dialogue histories.

4.2 Baselines

We compare AdaMemPIP against representative baselines from recent memory-agent work.

- *Full Text*: It answers directly from the full dialogue history without a memory module.
- *Naive RAG*: It stores dialogue content as retrievable text and applies standard retrieval-augmented generation (Lewis et al., 2020).
- *LangMem*: It is a memory-augmented agent design included in the LightMem comparison setting (Fang et al., 2025).
- *A-MEM*: It is a structured agentic memory framework for LLM agents (Xu et al., 2025).
- *MemoryOS*: It is a memory operating-system style framework reported in the LightMem comparison setting (Fang et al., 2025).
- *Mem0*: It is a scalable long-term memory framework (Chhikara et al., 2025).
- *LightMem*: It is a lightweight memory augmented generation framework and our strongest baseline (Fang et al., 2025).

These baselines cover a broad range of design choices, from direct context use to structured and lightweight memory systems. We follow the same baseline set used in the LightMem comparison. This breadth is important. If adaptive orchestration is useful, it should improve over both simple retrieval baselines and stronger memory systems.

4.3 Evaluation Metrics

We follow the evaluation protocol of Fang et al. (2025) and keep the benchmark setting aligned with prior work. On LongMemEval, we report accuracy for each question category and a weighted total score. On LoCoMo, we report overall accuracy following the standard benchmark metric.

4.4 Implementation Details

We use GPT-4o-mini and Qwen3-30B-A3B-Instruct-2507 as the backbone models of AdaMemPIP. We evaluate AdaMemPIP end to end on both

Table 1: Accuracy (%) results across question types on LongMemEval. The results of baselines in each category are taken from (Fang et al., 2025). Best scores within each backbone block are marked in **bold**. Second-best scores within each backbone block are marked with underlining. Tied best scores are all boldfaced. The **Total** column reports the weighted average over all categories using the number of examples in each category.

Method	Temporal (n=133)	Multi-Session (n=133)	Knowledge-Update (n=78)	Single-User (n=70)	Single-Assistant (n=56)	Single-Preference (n=30)	Total
GPT-4o-mini							
Full Text	31.58	45.45	76.92	87.14	89.29	36.67	56.89
Naive RAG	39.85	48.48	67.95	90.00	**98.21**	53.33	60.90
LangMem	15.79	20.30	66.67	60.00	46.43	60.00	37.20
A-MEM	47.36	48.87	64.11	**92.86**	96.43	46.67	62.20
MemoryOS	32.33	31.06	48.72	80.00	64.29	30.00	44.66
Mem0	40.15	46.21	70.12	81.43	41.07	60.00	53.51
LightMem	67.18	**71.74**	83.12	87.14	32.14	68.18	69.81
AdaMemPIP	**72.18**	67.67	**85.9**	88.57	58.90	**76.67**	**74.20**
Qwen3-30B-A3B-Instruct-2507							
Full Text	33.08	35.61	76.92	82.86	87.50	50.00	54.67
Naive RAG	36.84	47.73	65.38	**91.43**	**98.21**	70.00	60.69
LangMem	37.60	38.35	67.95	78.57	42.86	70.00	50.80
A-MEM	51.88	51.12	76.93	90.00	96.43	40.00	65.20
MemoryOS	28.57	36.84	61.54	72.86	92.86	33.33	49.60
Mem0	41.94	28.13	28.57	55.32	26.09	81.82	38.67
LightMem	54.20	51.91	66.67	80.00	31.25	80.00	58.13
AdaMemPIP	**66.92**	**58.33**	**78.21**	**91.43**	41.82	**100.00**	**69.08**

benchmarks. For each example, we replay the preceding dialogue records in chronological order. Each record updates the memory state through the storage path before the final benchmark query is asked. The usage path then plans retrieval, selects one reader, and passes only the selected evidence to the backbone model. For each retrieval call, we keep the top-10 returned memory items. All routing is prompt-based and zero-shot. We do not fine-tune the controller. Final predictions are scored with the official benchmark metrics, using GPT-4o-mini as the judge model for accuracy evaluation.

4.5 Main Results

Table 1 shows that adaptive orchestration improves long-term memory quality. AdaMemPIP achieves the best total score on both backbones. On GPT-4o-mini, it improves over LightMem from 69.81 to 74.2 (+4.4). On Qwen3-30B-A3B-Instruct-2507, it improves from 58.13 to 69.08 (+11.0). The strongest improvements appear on categories that need more than one memory action. Temporal, Knowledge-Update, and Single-Preference all benefit from better control over extraction, retrieval, and evidence presentation. The weakest category remains Single-Assistant. AdaMemPIP improves over LightMem there, but still trails retrieval-heavy baselines such as Naive RAG and A-MEM. This shows that assistant-grounded recall remains a real challenge. Overall, adaptive orchestration gives a better balance between recall, update, and reuse than a fixed pipeline.

4.6 Results on LoCoMo

To test generalization beyond LongMemEval, we compare methods on LoCoMo (Maharana et al., 2024). For methods with multiple reported variants, we use the strongest reported result for each backbone. We therefore treat this as a cross-paper generalization check rather than a strict same-run comparison. Figure 3 shows a mixed but encouraging pattern. AdaMemPIP is the best method on GPT-4o-mini, improving over LightMem from 72.99 to 77.86. On Qwen3-30B, it is competitive but not best. This is still useful evidence. It suggests that adaptive control transfers beyond one benchmark, but its benefit still depends on the backbone model.

4.7 Ablation Study

We next study how each adaptive component contributes to the final performance. We use the GPT-4o-mini setting and keep all modules adaptive by

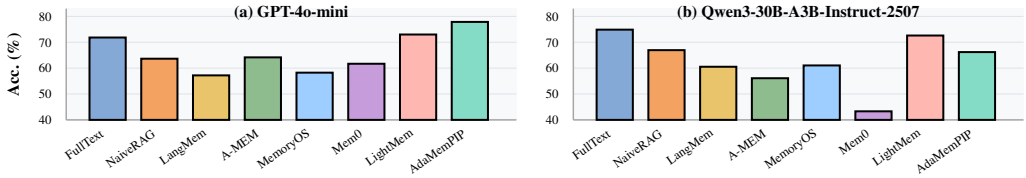

Figure 3: Accuracy comparison on LoCoMo under two backbone models. For methods with multiple reported variants, we use the best reported LoCoMo accuracy for each backbone.

default. In each ablation, we replace one component family with a single fixed choice while leaving the rest unchanged. For extraction, we test fact-only, preference-only, and causal-only variants. For retrieval, we test vector-only and keyword-only variants. For reading, we test *FlatReader*, *AssistantReader*, and *TimelineReader*. This experiment isolates the role of each adaptive component. Table 2 shows that all three component families matter, but in different ways. The extractor ablations cause the largest collapse. Fact-only extraction remains reasonably strong, but preference-only and causal-only extraction fail badly. This shows that the system cannot rely on one memory type across all tasks. The retriever ablations reveal a different pattern. Vector retrieval is a strong default choice. Keyword retrieval alone is much weaker. However, the full system is still better than vector-only retrieval. The gain therefore comes from adaptive retrieval, not from picking one strong default. The reader ablations provide the clearest evidence for step-level adaptivity. No single reader dominates across all categories. FlatReader helps on broad information questions. AssistantReader helps on assistant-centered questions. TimelineReader helps on temporal reasoning. Once the reader is fixed, the total score drops. This directly supports our main claim that evidence organization should be adaptive rather than static.

4.8 Activation Statistics

To better understand how AdaMemPIP behaves in practice, we analyze the activation frequency of each memory operation. Some modules can be activated jointly, so we use grouped bar charts.

Figure 4 shows that the controller does not collapse to one path. Fact extraction reaches 100%, partly because factual memory is broadly useful and partly because the routing prompt treats fact memory as the default library. Preference and causal extraction are less frequent, but still common enough to show that the controller uses multiple memory types when needed.

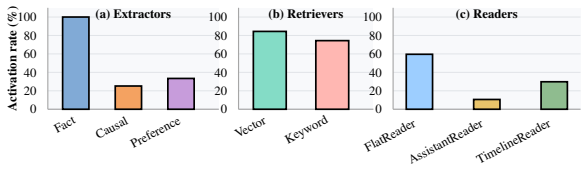

Figure 4: Activation frequency of different memory operations. Extractors and retrievers can be co-activated, so their percentages do not need to sum to 100%.

For retrieval, vector retrieval is the dominant choice, while keyword retrieval is also used frequently. This is consistent with the ablation results. The reader statistics are more selective. FlatReader is the most common strategy. TimelineReader is also used regularly. AssistantReader is much rarer. This matches the benchmark composition: most queries need only lightweight evidence packing, while a smaller subset needs temporal ordering or assistant-side grounding.

4.9 Memory Construction Cost

We finally examine the token cost of memory construction. We measure the total summary-token usage for each example by summing summary input and output tokens. This metric isolates memory construction itself. It does not include final answer generation or the small routing prompts. Figure 5 shows that the gains do not come with an unacceptable construction cost. MemoryOS is by far the most expensive. Mem0 and A-MEM are also much heavier than lightweight methods. LightMem has the lowest cost. AdaMemPIP is more expensive than LightMem, but remains much cheaper than the heavier structured-memory baselines. This suggests that the gains do not come from simply spending more tokens on every example.

4.10 Case Study

To illustrate how adaptive orchestration works in practice, we present two cases from LongMemEval. The first requires temporal reasoning over two distant events. The second asks for a detail from a pre-

Table 2: Ablation results on LongMemEval with GPT-4o-mini. Each ablation fixes one component choice while keeping the remaining modules adaptive.

Variant	Temporal	Multi-Session	Knowledge-Update	Single-User	Single-Assistant	Single-Preference	Total
AdaMemPIP	72.2	67.7	85.9	88.6	58.9	76.7	74.2
Extractor Ablations							
Fact only	69.2	60.2	85.9	88.6	58.9	76.7	71.4
Preference only	6.0	5.3	6.4	5.7	14.3	6.7	6.8
Causal only	24.1	9.0	15.4	8.6	23.2	26.7	16.6
Retriever Ablations							
Vector only	68.4	68.4	84.6	88.6	55.4	76.7	72.8
Keyword only	42.1	36.8	65.4	54.3	44.6	26.7	45.4
Reader Ablations							
FlatReader only	69.9	60.9	87.2	88.6	30.4	73.3	68.8
AssistantReader only	56.4	57.9	76.9	81.4	66.1	66.7	65.2
TimelineReader only	72.9	66.9	83.3	85.7	33.9	76.7	70.6

Table 3: Case study examples from LongMemEval. The two cases show that AdaMemPIP chooses different retrieval and reader paths for different query types.

	Query	Adaptive selection	Retrieved evidence	Reasoning and answer
Case 1 Temporal reasoning	How many days passed between my visit to MoMA and the "Ancient Civilizations" exhibit at the Metropolitan Museum of Art?	Fact/vector Preference/vector Causal/vector *TimelineReader*	Visited Museum of Modern Art on Jan 8, 2023. Attended the Metropolitan Museum exhibit on Jan 15, 2023. Related causal memory supports event linkage.	Vector retrieval expands semantic recall to recover both distant events. TimelineReader then sorts them into a timeline, so the generator can directly compute the gap. **Answer:** 7 days.
Case 2 Single-session assistant	Making a classic French omelette ... how many eggs did you say we need?	Fact/keyword keyword = omelette Fact/vector Preference/vector Causal/vector *AssistantReader*	Keyword retrieval locks onto the omelette discussion. Assistant-side evidence contains the earlier reply: "2–3 eggs."	The query explicitly asks what the assistant said before. AssistantReader therefore surfaces the original assistant utterance instead of only user facts. **Answer:** 2–3 eggs.

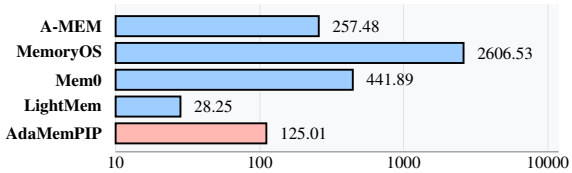

Figure 5: Memory construction cost measured by total summary tokens per example. Lower is better.

vious assistant response. The two cases stress different memory needs and different memory paths. Table 3 highlights the main advantage of AdaMemPIP. The system does not rely on one default retrieval route or one default reader. In the temporal case, it benefits from broader semantic recall and a timeline-style view. In the assistant-centered case, it benefits from precise lexical matching and direct recovery of assistant utterances. This is the practical value of adaptive orchestration. The gain does not come only from storing more memory or retrieving more text. It comes from matching the memory process to the question.

5 Conclusion and Future Work

This paper studies adaptive orchestration for agent memory. We argue that fixed memory pipelines are too rigid for long-term interactive settings. We introduce AdaMemPIP, a prompt-based framework that selects memory operations at the step level. Across LongMemEval and LoCoMo, the results show that this design improves memory quality while keeping memory construction cost practical. The additional analyses show that the gains come from adaptive coordination across extraction, retrieval, and reading. Future work can extend the function library with richer stores and more specialized readers. It can also improve assistant-centered grounding and explore stronger controllers.

Limitations

AdaMemPIP has several limitations. First, the controller is prompt-based, so routing may be sensitive to prompt wording and backbone behavior. A wrong route can skip a useful memory library or choose an unsuitable reader. Second, our efficiency analysis follows prior work and focuses on memory construction cost. It does not measure full serving cost, which also includes routing prompts, retrieval latency, answer generation, memory size, and the chosen backbone model. Third, the current function library is small. We use three memory types, two retrieval methods, and three readers. Richer applications may need more specialized stores, multimodal memories, deletion or conflict-resolution operators, and stronger temporal reasoning. Finally, our main experiments focus on long-term dialogue memory benchmarks. These benchmarks are useful, but they cannot capture all real user settings, such as noisy conversations, privacy-sensitive memory, rapidly changing profiles, or agents that run for months.

References

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

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. 2025. Lightmem: Lightweight and efficient memory-augmented generation. *CoRR*, abs/2510.18866.

Yannis Katsis, Sara Rosenthal, Kshitij Fadnis, Chulaka Gunasekara, Young-Suk Lee, Lucian Popa, Vraj Shah, Huaiyu Zhu, Danish Contractor, and Marina Danilevsky. 2025. mtrag: A multi-turn conversational benchmark for evaluating retrieval-augmented generation systems. *Trans. Assoc. Comput. Linguistics*, 13:784–808.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*.

Junfeng Lu and Yueyan Li. 2025. Dynamic affective memory management for personalized llm agents. *CoRR*, abs/2510.27418.

Weitao Ma, Xiaocheng Feng, Lei Huang, Xiachong Feng, Zhanyu Ma, Jun Xu, Jiuchong Gao, Jinghua Hao, Renqing He, and Bing Qin. 2026. Fine-mem: Fine-grained feedback alignment for long-horizon memory management. *CoRR*, abs/2601.08435.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. 2024. Evaluating very long-term conversational memory of llm agents. In *ACL*.

Niloofar Mireshghallah, Neal Mangaokar, Narine Kokhlikyan, Arman Zharmagambetov, Manzil Zaheer, Saeed Mahloujifar, and Kamalika Chaudhuri. 2025. Cimemories: A compositional benchmark for contextual integrity of persistent memory in llms. *CoRR*, abs/2511.14937.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. 2023. Memgpt: Towards llms as operating systems. *CoRR*, abs/2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *UIST*.

Rana Salama, Jason Cai, Michelle Yuan, Anna Currey, Monica Sunkara, Yi Zhang, and Yassine Benajiba. 2025. Meminsight: Autonomous memory augmentation for llm agents. In *EMNLP*.

Miao Su, Yucan Guo, Zhongni Hou, Long Bai, Zixuan Li, Yufei Zhang, Guojun Yin, Wei Lin, Xiaolong Jin, Jiafeng Guo, and Xueqi Cheng. 2026. Beyond dialogue time: Temporal semantic memory for personalized llm agents. *CoRR*, abs/2601.07468.

Haoran Tan, Zeyu Zhang, Chen Ma, Xu Chen, Quanyu Dai, and Zhenhua Dong. 2025. Membench: Towards more comprehensive evaluation on the memory of llm-based agents. In *ACL Findings*.

Yufei Tao, Adam Hiatt, Erik Haake, Antonie J. Jetter, and Ameeta Agrawal. 2024. When context leads but parametric memory follows in large language models. In *EMNLP*.

Zheng Wang, Zhongyang Li, Zeren Jiang, Dandan Tu, and Wei Shi. 2024. Crafting personalized agents through retrieval-augmented generation on editable memory graphs. In *EMNLP*.

Rebecca Westhäußer, Wolfgang Minker, and Sebastian Zepf. 2025. Enabling personalized long-term interactions in llm-based agents through persistent memory and user profiles. *CoRR*, abs/2510.07925.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. 2025. Longmemeval: Benchmarking chat assistants on long-term interactive memory. In *ICLR*.

Xinle Wu, Rui Zhang, Mustafa Anis Hussain, and Yao Lu. 2026. Towards autonomous memory agents. *CoRR*, abs/2602.22406.

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

Kai Zhang, Lizhi Qing, Yangyang Kang, and Xiaozhong Liu. 2024. Personalized llm response generation with parameterized memory injection. *CoRR*, abs/2404.03565.

Qian Zhao, Zhuo Sun, Bin Guo, and Zhiwen Yu. 2025. Memocue: Empowering llm-based agents for human memory recall via strategy-guided querying. In *ECAI*.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2024. Memorybank: Enhancing large language models with long-term memory. In *AAAI*.

A Prompt Templates for Routing

This appendix presents the retrieval-routing prompt template used by the controller. Please see the code for all detailed prompts.

Table 4: Retrieval-routing prompt used to select memory libraries and search methods for sub-queries.

Prompt Content	
Role. You route sub-queries for a personal-memory RAG system. The retriever has only two methods. *vector* is used for paraphrases, broad semantic lookup, fuzzy user intent, and preferences. *keyword* is used for one specific noun/entity phrase, which must be provided explicitly.	
Memory libraries. *fact*: what/when/where/who/how many, events, dates, order, and updated facts. *preference*: likes, dislikes, habits, and stable opinions. *causal*: reasons, consequences, and causal chains.	
Output schema. Return exactly one JSON object and no markdown. Preferred schema: `{"queries":[{"library":"fact","method":"vector","keywords":[]}]}`. Valid library values are `fact`, `preference`, `causal`, and `all`. Valid method values are `vector` and `keyword`. Keyword search must use exactly one noun/entity phrase. The whole plan may contain at most two keyword query objects.	
Routing rules. Always include fact memory at minimum. Prefer the smallest library set that still covers the sub-query. Prefer vector when wording may vary. Prefer keyword only for exact noun/entity matches. Do not use keyword for control words or time windows. Temporal, counting, and ordering questions usually use fact memory. Pure preference questions use preference memory. If unsure, use vector over all.	
Representative examples.	

Temporal ordering	`{"queries":[{"library":"fact","method":"keyword","keywords":["job interview"]},`
	`{"library":"fact","method":"keyword","keywords":["vacation trip"]},`
	`{"library":"fact","method":"vector","keywords":[]}]}`

Single entity + date	`{"queries":[{"library":"fact","method":"keyword","keywords":["bike"]},` `{"library":"fact","method":"vector","keywords":[]}]}`
Preference	`{"queries":[{"library":"fact","method":"vector","keywords":[]},` `{"library":"preference","method":"vector","keywords":[]}]}`
Causal	`{"queries":[{"library":"fact","method":"keyword","keywords":["cancel subscription"]},` `{"library":"causal","method":"vector","keywords":[]}]}`