AGENT MEMORY FOR LONG-HORIZON LANGUAGE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-running language agents repeatedly draw on growing interaction histories. Memory construction organizes history into compact context, but summaries formed before future queries are known may omit needed details. More elaborate construction and maintenance also incur additional model calls, repeated encoding, and persistent state storage. We revisit the role of constructed memory: it should provide directly readable, organized context and help locate relevant sources, rather than serve as the sole carrier of historical evidence. We introduce FlashMem, which co-designs agent-side memory representation and server-side key–value (KV) cache reuse. On the agent side, one-pass construction produces compact, source-linked memory. Answering combines retrieved memory with a small amount of original evidence from its linked sources. On the server side, FlashMem reuses recurring context across construction calls and retains KV states for compact memory rather than the full history across queries. Across three datasets, FLASHMEM achieves relative F1 gains of 17.4%, 33.1%, and 62.3%, while reducing inference costs by 97.1%. FLASHMEM's server-side optimizations reduce time to first token (TTFT) by 66.8%, selection overhead for KV recomputation by 99.3%, and long-term KV storage costs by 84.9–93.8%. The framework also accelerates other agent memory methods. We will release our code after anonymous review.

1 INTRODUCTION

Long-running language agents use past facts, events, and state changes in later interactions. Although large language models have made substantial progress in understanding and reasoning (Achiam et al., 2023; Grattafiori et al., 2024; Yang et al., 2025), bounded context windows and the cost of long-context processing limit their direct use of extended histories (Packer et al., 2023; Chhikara et al., 2025). Retrieval-augmented generation (RAG) provides access to past interactions by retrieving relevant passages at query time (Packer et al., 2023; Wu et al., 2024). Agent memory systems extend this paradigm with an explicit construction stage that organizes history into persistent, retrievable representations (Zhong et al., 2024; Xu et al., 2026b; Chhikara et al., 2025). Their workflow therefore comprises two related phases: **construction**, which organizes interaction histories, and **inference**, which retrieves and uses this information to answer queries.

These phases operate under different information constraints. Construction organizes history before queries are known, whereas inference may depend on details of a specific event. If constructed memory is the sole evidence source, details omitted during construction are unavailable at query time. Additional extraction, linking, consolidation, and updating can retain more information, but increase construction and maintenance costs (Zhang et al., 2026; Chen et al., 2026; Omri et al., 2026). Relying on raw history does not remove the need for organization: information may be scattered across exchanges and mixed with narrative. The central question is therefore *what to organize into memory in advance and what to retrieve from the original history once a query is known*.

We argue that constructed memory should organize history rather than replace it. Compact memory consolidates dispersed information into coherent descriptions of facts and events that the inference model can read directly, while source links help locate the underlying evidence. The original history retains details omitted during construction for query-specific retrieval. This division lets construction focus on compact representations without encoding every detail future queries might need.

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.

This division also changes which computational states need to persist on the server. Retaining history as text preserves access to evidence; retaining its key–value (KV) cache avoids repeated encoding. These two forms of persistence need not cover the same content. When compact memory supplies recurring context and raw details are retrieved on demand, persistent KV reuse can focus on memory rather than the full history. Yet compaction alone does not eliminate execution overhead. Recurring content in construction and inference may appear under different prefixes or orderings, limiting conventional prefix caching (Ye et al., 2026; Hu et al., 2024). Position-independent caching (PIC) adapts cached KV states to new contexts through selective recomputation (Yao et al., 2025; Hu et al., 2024; Yang et al., 2026). As cached contexts shorten, selecting positions for recomputation can take a larger share of online execution time. Memory representation determines *what to retain for reuse*; the reuse mechanism determines *how efficiently to adapt them to new contexts*, motivating co-design.

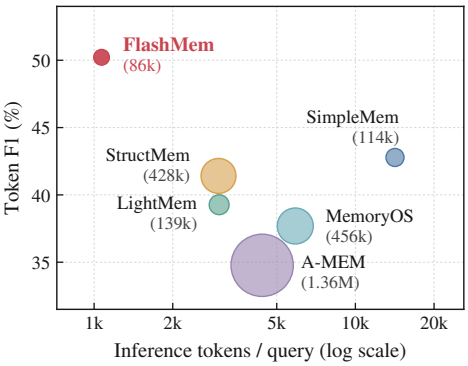

Figure 1: **Agent quality–cost Pareto frontier on LoCoMo.** Results use Qwen3-8B. Inference cost includes retrieval and answering; bubble area and parenthetical labels show construction cost. All costs count full input (including system prompts) and output tokens.

We introduce FlashMem, which co-designs agent-side memory representation and server-side KV reuse. On the agent side, FlashMem processes each new history segment once, consolidating repeated or complementary descriptions into self-contained entries with source identifiers. It appends these entries without retrieving or rewriting existing memory. At query time, it retrieves compact entries, then raw evidence from their linked source regions. Both views enter a single answer-generation call: the memory entries supply organized context, and short source windows add details relevant to the query. Figure 1 shows the improvements in answer quality and token cost.

On the server side, FlashMem accelerates construction and inference through KV reuse. During construction, it reuses KV states for recurring context across calls. During inference, it retains compact memory KV across queries. To reduce selection overhead across phases, a small model selects token positions for recomputation. The target model recomputes the KV states and reuses cached states to generate the output. The agent-side design reduces the context processed and persistently cached; server-side reuse reduces repeated computation on that context. Our contributions are as follows:

- We identify a division of labor for long-term memory: constructed memory provides compact, directly readable context and guides evidence retrieval, while the original history retains query-specific details. This division lets persistent KV reuse focus on compact memory.
- We introduce **FlashMem**, which combines one-pass memory construction with source-linked evidence retrieval. With Qwen3-8B, it improves F1 by **17.4%/33.1%/62.3%** over the best baseline on LoCoMo, LongMemEval-S, and HaluMem, respectively. Relative to SimpleMem, it reduces construction and inference token costs by up to **57.2%** and **93.4%**, respectively.
- We build a lightweight KV-reuse pipeline for memory construction and inference in server-side scenarios. At 30% recomputation, it reduces TTFT by 30.2%/41.0%/31.4% over existing baseline across models. Integrated with FlashMem, it delivers a **2.71**× median-TTFT speedup. The same pipeline also generalizes to five existing agent-memory systems, achieving 2.59–2.98× speedups.

2 Related Work

Agent memory. Agent memory retains and organizes past interactions outside an LLM's active context for later retrieval and reasoning (Xu et al., 2026b; Fang et al., 2026). Mem0 extracts facts, while StructMem preserves event-level bindings and cross-event relations (Chhikara et al., 2025; Xu et al., 2026a). A-MEM enriches and links notes, and MemoryOS consolidates information across memory tiers; these structures support recall but require repeated model processing during insertion and updates (Xu et al., 2026b; Kang et al., 2025). LightMem filters dialogue, groups related content, and defers long-term updates to reduce construction overhead (Fang et al., 2026). SimpleMem combines structured compression and recursive consolidation with query-specific retrieval planning (Liu et al., 2026). ReadAgent reopens source pages, and xMemory expands toward episodes and raw

messages, preserving access to details beyond compact records (Lee et al., 2024; Hu et al., 2026). However, recent studies increasingly question whether expensive memory construction is necessary for conversational memory (Zhou et al., 2026; Li et al., 2026; Wang, 2026; Xu et al., 2026c).

Position-independent caching. PIC enables chunk-level KV reuse across requests despite changes in a chunk's prefix or position (Hu et al., 2024). In retrieval-augmented generation, this avoids re-encoding recurring passages, but independently cached states lack cross-chunk interactions. CacheBlend progressively selects tokens for recomputation using KV deviations (Yao et al., 2025); EPIC's LegoLink recomputes chunk-initial tokens to mitigate attention sinks from independent encoding (Hu et al., 2024). FusionRAG precomputes cross-attention with similar chunks offline and uses target-model query attention for recomputation selection (Wang et al., 2026a). Recent work extends reuse to agent pipelines: KVCOMM estimates prefix-induced offsets from cached anchors to share overlapping context across agents (Ye et al., 2026), while RelayCaching reuses upstream decoding states during downstream prefill with selective contextual correction (Geng et al., 2026). In long-term agent memory, however, histories are long and constructed summaries often remain comparable in length, so directly applying RAG PIC can incur substantial persistent KV storage.

3 PRELIMINARY

Agent memory pipeline. We abstract the retrieval-based memory workflow considered here into construction, query-dependent reading, and answer generation. Given history $\mathcal{H}$, construction G produces a persistent memory store $\mathcal{D}$. For query q, Read retrieves and assembles evidence E_q under context budget β, and inference model F_θ generates answer a:

$$\mathcal{D} = G(\mathcal{H}), \quad E_q = \mathrm{Read}(q, \mathcal{D}; \beta), \quad a = F_\theta(q, E_q) \tag{1}$$

Here, θ denotes the target model's parameters.

Position-independent KV reuse. Let $X = C_1 \| \cdots \| C_m \| T$ contain reusable chunks C_i and request-specific tokens T, where $\|$ denotes concatenation. Position-independent caching (PIC) reuses chunk KV states under new prefixes or orderings (Hu et al., 2024; Yao et al., 2025; Yang et al., 2026). At layer ℓ, let $\widetilde{\mathrm{KV}}_\theta^{(\ell)}$ denote cached pairs assembled in prompt order and aligned to current positions. RoPE alignment adjusts key rotations while preserving values (Appendix D.2); it does not recover interactions with new preceding context, so direct reuse generally differs from full prefill.

Selective recomputation updates only part of this cache. Let $\mathcal{U}$ contain uncached or mandatory positions and $\mathcal{R}$ the cached positions eligible for recomputation. A policy controlled by $\rho \in [0, 1]$ selects $I \subseteq \mathcal{R}$, and the target processes $\mathcal{S} = \mathcal{U} \cup I$. Section 5.2 defines the selector-token budget and its mapping to I. Under the fixed-set execution, the cache at each layer is formed as follows:

$$\widehat{\mathrm{KV}}_\theta^{(\ell)}[i] = \begin{cases} \mathrm{KV}_{\theta,\mathrm{new}}^{(\ell)}[i] & i \in \mathcal{S} \\ \widetilde{\mathrm{KV}}_\theta^{(\ell)}[i] & i \notin \mathcal{S} \end{cases} \tag{2}$$

New pairs derive from current target states, initialized from first-layer embeddings. For one head (index omitted), the updated pairs $(\widehat{K}_\theta^{(\ell)}, \widehat{V}_\theta^{(\ell)})$ yield the following active-position output:

$$O_\theta^{(\ell)}[\mathcal{S}] = \mathrm{softmax}_{\mathrm{row}}\left(\frac{Q_\theta^{(\ell)}[\mathcal{S}](\widehat{K}_\theta^{(\ell)})^\top}{\sqrt{d_\theta}} + M_\mathcal{S} \right) \widehat{V}_\theta^{(\ell)} \tag{3}$$

Here, d_θ is the head dimension and $M_\mathcal{S}$ enforces causality using prompt positions. Active queries read all visible KV rows, including unrecomputed ones; only their hidden states advance through the target layers. The final prompt position belongs to $\mathcal{U}$ for autoregressive decoding.

4 MOTIVATION

Agent memory is constructed before queries are known, creating an asymmetry between construction and inference. Construction must decide what information to preserve without knowing which details will matter, whereas inference can access history with a specific query in hand. A single constructed representation is difficult to make both compact and sufficient for every query, while processing and

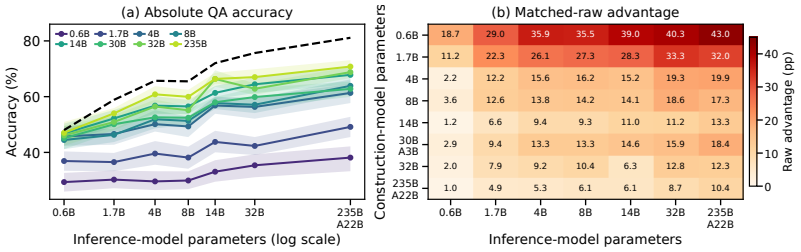

Figure 2: **Summary-based QA trails matched raw context across model scales.** LoCoMo QA with Qwen3 models and matched source sessions. (a) Absolute accuracy; colored curves use different construction models and the dashed line is the raw-context ceiling. (b) Raw-context advantage over summaries (pp). Shading shows 95% paired cluster-bootstrap intervals.

retaining the full interaction history as reusable model state is unnecessarily expensive. This raises a basic question: what should be organized into memory in advance, and what should remain in the original history for query-specific access? The answer also determines what content must persist as reusable model state on the server. We first examine the role of constructed memory relative to raw history, and then study how this representation should interact with KV reuse.

4.1 Constructed Memory as Compact Context

Observation 1: Query-blind summaries can omit evidence needed by future queries. Figure 2 cross-evaluates construction and answering models across scales. For each question, we compare QA accuracy using the source sessions describing the relevant event with that using summaries constructed from the same sessions, thereby isolating the effect of memory representation from retrieval quality. Summary-based accuracy generally improves as either model scales up (Figure 2(a)). Yet, for the same answering model, matched raw context typically remains more accurate, and this advantage sometimes widens with stronger answering models (Figure 2(b)). This persistent gap is consistent with useful details being omitted during construction: stronger answering models can better exploit the original evidence, but cannot recover content absent from the summary. Scaling construction or answering alone does not eliminate the need for access to original evidence.

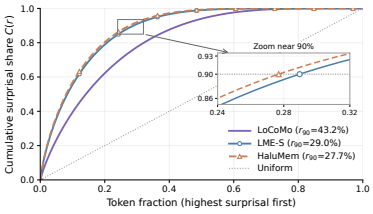

Figure 3: **Surprisal concentrates in a fraction of history.** Qwen3-8B estimates averaged across histories; shading spans the interquartile range. r_{90} gives the token fraction carrying 90% of surprisal on the mean curve.

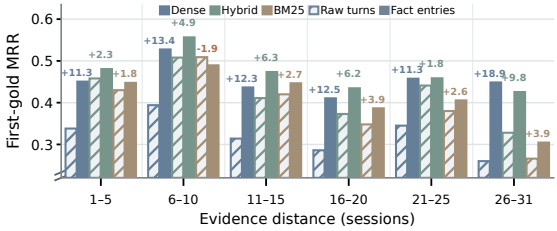

Figure 4: **Constructed entries show higher MRR in most groups.** First-gold MRR on LoCoMo by evidence distance (sessions back to the earliest gold evidence). Labels give fact-entry minus raw-turn MRR.

Observation 2: Constructed memory can help localize relevant history by organizing information. Retaining the original history does not make memory construction unnecessary. Observation 1 controls for retrieval by providing the relevant source context; in practice, however, those sources must first be located within a long interaction history. Following the token-level self-information measure of Li et al. (2023), Figure 3 shows that information is concentrated in only a fraction of history tokens, leaving substantial background content over which retrieval must search. The matched-source setting in Observation 1 therefore isolates representation quality but understates the retrieval burden faced in practice. We further compare source localization using raw turns and constructed fact entries. Figure 4 groups questions by evidence distance under BM25, dense, and hybrid retrieval. Constructed fact entries achieve higher first-gold mean reciprocal rank (MRR; Appendix C.1) in most groups, including evidence farther back in the history. Importantly, helping locate relevant sources is distinct from independently preserving every detail needed for answering. Constructed memory and original history are therefore complementary: the former organizes dispersed information and helps localize relevant sources, while the latter preserves the original evidence needed for query-specific access.

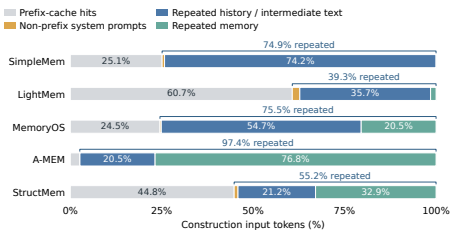

Figure 5: **Repeated inputs beyond shared prefixes.** LoCoMo construction. Brackets: non-prefix repeats; gray: prefix-cache hits.

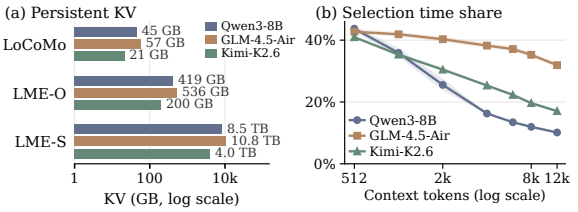

Figure 6: **Storage and selection constrain KV reuse.** (a) Full-history BF16 KV estimates (compressed MLA for Kimi-K2.6). (b) FusionRAG selection's share of TTFT.

Insight 1. ***Construct memory for compact context while preserving access to original evidence.*** *Constructed memory provides organized context for retrieval and answering, while the original history remains available for query-specific evidence access. This division allows the constructed representation to remain compact without making it the sole source of evidence.*

4.2 Opportunities and Constraints of KV Reuse

Agent-memory workloads expose reuse opportunities across construction and inference, but incur different costs. During construction, recurring instructions, history, intermediate outputs, and memory entries are repeatedly encoded across calls, creating repeated prefill. During inference, memory is repeatedly accessed by queries, so reusing its computation requires KV states to persist across requests, introducing long-term storage cost. Position-independent KV reuse also incurs execution overhead: it must determine which cached positions require correction and then partially recompute states under the new context. We examine these reuse opportunities and PIC execution costs in turn.

Observation 3: Memory workloads expose distinct reuse opportunities across construction and inference. Figure 5 profiles construction inputs across five representative memory systems. Beyond conventional prefix-cache hits, 39.3–97.4% of processed input tokens recur as instructions, history, intermediate outputs, or memory entries. Because these segments often appear under different prefixes, conventional prefix caching cannot reuse them. This motivates position-independent KV reuse across construction calls. Inference-time reuse introduces a different constraint: reusable KV states must persist across queries. Figure 6(a) shows that retaining complete histories as KV states incurs substantial storage that grows with history length. The original history can instead remain accessible as text, while compact memory provides a smaller target for persistent KV reuse.

Observation 4: Compact memory reduces KV storage but leaves substantial selection overhead. PIC execution lies on the critical path to the first token: query-guided reuse first selects cached positions for correction and then partially re-prefills those positions with the target model. Figure 6(b) shows that selection accounts for a substantial fraction of TTFT, and this fraction increases as the reused context becomes shorter. As partial re-prefill shrinks with shorter reusable context, selection becomes an increasingly prominent part of the remaining execution.

Together, these observations reveal a coupled design problem. Compacting the reusable representation reduces persistent KV storage, but cannot eliminate the selection overhead of PIC; conversely, PIC can reduce repeated computation for a given context, but does not decide what historical content should become long-lived model state. Agent memory provides this upstream design choice, allowing compact memory to remain resident as reusable KV while original history stays available as text for on-demand evidence access.

Insight 2. ***Co-design memory representation with KV reuse.*** *Memory representation determines what should be retained for reuse, while PIC determines how efficiently that state can be adapted to new contexts. The two layers should therefore jointly reduce persistent state and online reuse overhead.*

5 Method

FlashMem combines both insights: it constructs and reads compact memory, then adds evidence from linked sources. It persistently caches memory states and processes raw content as needed.

Section 5.1 defines construction and reading in Equation (1); Section 5.2 defines low-cost selection of recomputation positions in Equation (2).

5.1 Compact Memory with Complementary Retrieval

Compression can omit details needed by future queries, while reading entire original records introduces irrelevant background. Following Insight 1, FlashMem combines two evidence views: compact memory summaries supply organized context, and retrieval within their associated source regions supplies original details. To instantiate Equation 1, construction G retains $\mathcal{D} = (\mathcal{M}, \mathcal{H})$, where $\mathcal{M}$ contains compact summaries. We implement Read through retrieval and reranking R, summary rendering A_{m}, and query-guided raw-window assembly A_{h}. Let S_q be the complete summaries selected from $R(q, \mathcal{M})$ with target budget β_{m}, and let $\mathrm{Src}(m)$ denote the original source units cited by summary m. Their union defines the source region $\mathcal{H}_q = \bigcup_{m \in S_q} \mathrm{Src}(m) \subseteq \mathcal{H}$ for complementary retrieval. With $\beta = (\beta_{\mathrm{m}}, \beta_{\mathrm{h}})$, the read operation assembles evidence as follows:

$$E_q = \mathrm{Read}(q, \mathcal{D}; \beta) = A_{\mathrm{m}}(S_q) \| A_{\mathrm{h}}(q, R(q, \mathcal{H}_q); \beta_{\mathrm{h}}) \tag{4}$$

Here, β_{m} controls summary selection, and β_{h} controls raw-window assembly.

Construction G: one-pass consolidation. We split sessions into chronological segments and use one construction-model generation per segment to produce retrievable items. The model is instructed to consolidate repeated or complementary descriptions, keep requests with answers, and record changes with temporal context. This reduces repeated background while keeping related information together. Each item records its source identifiers for subsequent source retrieval. Items are appended without retrieving or rewriting existing memory, avoiding repeated maintenance.

Retrieval R: summaries followed by source details. We first apply hybrid retrieval and reranking to $\mathcal{M}$ and select complete summaries into S_q. Their source identifiers locate and deduplicate the original units forming $\mathcal{H}_q$. We then apply hybrid retrieval and reranking with the same query q within $\mathcal{H}_q$ to select raw candidates. The summaries provide organized context and locate relevant history regions; retrieval within those regions supplies details that the summaries may omit.

Evidence assembly: complete items and local windows. Summary selection retains complete items in relevance order, using β_{m} as its target budget, and A_{m} renders the selected set S_q. Keeping an item intact preserves conditions and update relationships that sentence-level trimming could separate. Raw evidence, however, may occupy only a small part of a source. A_{h} scores spans against q, ranks sources by their best span, and first admits fitting seed spans across sources. It then uses the remaining β_{h} to expand each window with its best-scoring feasible adjacent spans. This gives multiple sources a chance to enter the context before expanding individual windows. Budget checks count fully rendered inference-model tokens, including source headers, and selected windows are copied verbatim. The complete procedure appears in Appendix A.6.

We order each evidence view chronologically to form E_q. The inference model then generates $a = F_\theta(q, E_q)$ in one call, completing the workflow in Equation 1. Each answer combines compact summaries with a small amount of raw evidence retrieved for that query. We therefore retain only summary KV states for PIC reuse across queries. The original history remains as text, and selected few raw evidence receive full prefill without persistent KV caching. This reduces persistent KV storage while preserving access to original details for answering.

5.2 KV Reuse with a Lightweight Selector

Reusing a summary in a new prompt can change its preceding context, so its cached KV states may need correction. Query-guided PIC selects tokens for recomputation based on task relevance (Wang et al., 2026a;b), but this selection adds online overhead (Yan et al., 2026). We introduce a lightweight selector to reduce this overhead. A small model F_ϕ selects token positions I, and the target model F_θ recomputes their KV states following Equation 2.

When F_θ builds the KV cache for each document chunk C_i, F_ϕ also prefills the same chunk once and caches only its keys from later layers $\mathcal{L}$. Online, F_ϕ encodes the short task text τ independently of cached context. For inference, τ is the user question; for construction, we phrase the task objective as a single-sentence question. We align task queries and cached keys with the task after the chunks.

At layer ℓ and head h, let $Q^{(\ell)}_{\phi,h} \in \mathbb{R}^{m_\mathrm{d} \times d_\phi}$ and $K^{(\ell)}_{\phi,h} \in \mathbb{R}^{n_\mathrm{d} \times d_\phi}$ denote task queries and cached keys. The average task-attention score of cached token j is computed as follows:

$$s_\phi(j) = \frac{1}{|\mathcal{L}| H_\phi m_\mathrm{d}} \sum_{\ell \in \mathcal{L}} \sum_{h=1}^{H_\phi} \sum_{u=1}^{m_\mathrm{d}} \left[\mathrm{softmax}_{\mathrm{row}} \left(\frac{Q^{(\ell)}_{\phi,h} (K^{(\ell)}_{\phi,h})^\top}{\sqrt{d_\phi}} \right) \right]_{u,j} \tag{5}$$

Here $m_\mathrm{d}, n_\mathrm{d}$ count task and cached selector tokens, H_ϕ counts query heads, and d_ϕ is their query–key dimension. Each row normalizes over all cached keys before eligibility filtering. Let $\mathcal{R}_\mathrm{d}$ be the selector-token positions whose source spans overlap eligible target positions in $\mathcal{R}$. We set $k = \lfloor \rho |\mathcal{R}_\mathrm{d}| \rfloor$ and take the top-k scores to obtain $I_\mathrm{d} \subseteq \mathcal{R}_\mathrm{d}$. We map their source spans to overlapping eligible target tokens to obtain $I \subseteq \mathcal{R}$ (Appendix D.3). Mandatory positions $\mathcal{U}$ do not count toward this budget; tokenizer boundaries can change the realized ratio $|I|/|\mathcal{R}|$. At each layer, the target updates $\mathcal{S} = \mathcal{U} \cup I$ and reuses the remaining KV rows.

Persistent keys amortize context encoding across requests, leaving a short task prefill, scoring, and token selection online. Table 1 quantifies these savings at 1K context: lightweight selection reduces selection time by 85.3–99.3% across three targets, with 1.6–4.37% estimated extra KV storage. Appendix D details the KV reuse procedure.

Table 1: Savings from replacing target-model selection with lightweight selection at 1K context.

Target model	TFLOPs↓	Selection time↓	Extra KV↑
Qwen3-8B	94.7%	85.3%	+2.08%
GLM-4.5-Air	97.0%	99.3%	+1.63%
Kimi-K2.6	98.8%	92.5%	+4.37%

6 Experiments

6.1 Experimental Setup

Datasets and models. We evaluate on LoCoMo (Maharana et al., 2024) (1,540 questions), LongMemEval-S (LME-S) (Wu et al., 2024) (500 questions), and HaluMem-Medium (Chen et al., 2025) (3,467 questions), using Qwen3-8B (Yang et al., 2025), GLM-4.5-Air (Zeng et al., 2025), and Kimi-K2.6 (Team et al., 2025). Component studies use Qwen3-8B unless noted and also include LME-O, which retains only the evidence sessions from LongMemEval.

Baselines and metrics. We compare with LightMem (Fang et al., 2026), A-MEM (Xu et al., 2026b), MemoryOS (Kang et al., 2025), SimpleMem (Liu et al., 2026), StructMem (Xu et al., 2026a), and raw-history RAG. We report token F1 and semantic accuracy judged by GLM-5.3, both on a 0–100 scale. Efficiency metrics cover token usage, executed prefill, latency, and KV storage. Token costs sum input and output unless reported separately. Appendix A provides details on more configurations.

6.2 Overall Performance

RQ1: Does FlashMem improve the end-to-end quality–cost trade-off? Table 2 shows consistent quality gains across datasets and model families. On Qwen3-8B, FlashMem achieves relative gains of **17.39%/33.12%/62.34%** in F1 and **10.48%/20.39%/3.05%** in accuracy over the strongest baseline for each metric on LoCoMo/LME-S/HaluMem, respectively, while lowering token usage relative to SimpleMem by up to **57.2%** during construction and **93.4%** during inference. The same accuracy advantage holds on every dataset with the larger GLM-4.5-Air and Kimi-K2.6 backbones.

Quality also remains competitive as KV reuse expands from inference alone (FlashMem†) to both construction and inference (FlashMem‡): Both variants achieve competitive answer quality across datasets and model backbones. Even with reuse in both phases, FlashMem‡ achieves 59.6% accuracy on LME-S, compared with 51.0% for the strongest baseline on Qwen3-8B. Table 4 shows that our 0.5B recomputation selector leads in accuracy and matches or exceeds both baselines in F1 on LoCoMo/GLM-4.5-Air at $\rho = 0.1/0.2/0.3$, with up to 4.9% higher accuracy than FusionRAG.

RQ2: How does the FlashMem design work? Table 3 compares four evidence configurations with Qwen3-8B and a fixed memory store: **S** uses compact memory only, **R** uses retrieved raw-history windows only, **S+R** retrieves raw evidence from the source context linked to selected memory entries, whereas **S+L** retrieves the two views independently across all three datasets. The advantage of **S+R** shows that the two views are most effective when coupled: compact memory localizes relevant information, while its source links recover the corresponding original evidence and details.

Table 2: Answer quality and mean per-sample token costs. In counts executed prefill tokens (including recomputation, excluding cache hits); Out counts generated tokens. Constr./Infer.: memory construction/retrieval and answering; k/M: $10^3/10^6$ tokens; –: unavailable. Bold/underline mark best/second-best quality scores. FlashMem uses no KV reuse; † reuses memory KV, and ‡ also reuses KV during construction.

Method	LoCoMo						LME-S						HaluMem					
	F1	ACC	Constr.		Infer.		F1	ACC	Constr.		Infer.		F1	ACC	Constr.		Infer.	
			In	Out	In	Out			In	Out	In	Out			In	Out	In	Out
Qwen Qwen3-8B																		
Raw RAG	32.64	49.14	–	–	3.6k	22	26.47	51.00	–	–	8.6k	30	23.87	65.56	–	–	3.6k	27
SimpleMem	42.78	50.78	41.9k	71.8k	16.2k	736	25.43	30.40	138.1k	65.0k	17.3k	773	23.87	53.85	420.9k	545k	5.0k	1.9k
LightMem	39.27	44.61	33.8k	46.7k	3.0k	14	25.89	51.00	76k	19k	1.1k	13	19.89	71.30	894.1k	159k	4.4k	19
MemoryOS	37.69	42.87	200.8k	107k	5.9k	21	30.56	47.60	513.3k	61.3k	8.5k	17	15.21	72.72	817.2k	298k	6.6k	248
A-MEM	34.77	41.23	1.22M	104k	4.4k	12	33.36	42.60	2.75M	186k	8.5k	13	14.20	44.45	3.22M	1.49M	2.5k	78
StructMem	41.41	45.39	217.9k	66.5k	3.0k	20	26.92	49.80	343k	74.8k	1.2k	12	19.92	68.68	5.64M	128k	4.6k	17
FlashMem	**50.22**	**56.10**	53.2k	11.5k	1.1k	10	**44.41**	**61.40**	136.3k	18.8k	1.6k	9	38.75	**74.94**	349k	64.5k	1.1k	61
FlashMem†	47.13	55.77	53.2k	11.5k	457	13	40.97	60.40	136.3k	18.8k	749	11	**40.29**	73.59	349k	64.4k	510	50
FlashMem‡	43.79	54.28	17.8k	7.4k	459	13	41.90	59.60	42.8k	15.0k	749	12	36.90	73.69	107k	50.5k	470	64
GLM GLM-4.5-Air																		
SimpleMem	**52.20**	65.78	41.1k	85.3k	12.7k	736	43.25	66.40	169.3k	130k	13.3k	829	33.54	69.02	298.7k	590k	2.0k	968
LightMem	41.99	45.00	33.7k	48.9k	2.1k	4	36.26	59.60	70.3k	20.3k	722	7	19.76	69.77	916.3k	209k	2.9k	9
MemoryOS	39.72	44.64	183.2k	96.6k	4.8k	7	30.08	57.80	481.9k	55.8k	8.1k	20	16.10	73.70	842.2k	341k	6.4k	277
A-MEM	38.67	42.86	2.66M	558k	9.3k	11	38.20	41.00	2.97M	265k	8.6k	13	21.55	47.59	3.88M	1.56M	3.8k	81
StructMem	44.31	49.81	207.3k	75.9k	5.0k	5	38.34	66.60	321.3k	90.7k	839	7	19.87	61.43	1.53M	233k	3.1k	9
FlashMem	51.07	**67.79**	47.1k	14.4k	1.0k	10	**48.45**	**75.20**	127.3k	26.2k	1.6k	64	**39.33**	80.58	306.9k	51k	1.1k	71
FlashMem†	50.46	64.04	47.1k	14.4k	477	10	47.03	74.60	127.3k	26.2k	763	67	39.03	**81.77**	306.9k	51k	515	74
FlashMem‡	50.34	62.07	15.3k	9.8k	464	8	47.24	73.60	39.4k	19.6k	729	65	38.06	80.53	93.3k	60.6k	516	75
Moonshot Kimi-K2.6																		
SimpleMem	53.56	73.77	38.2k	81.5k	13.0k	972	49.78	74.40	189.6k	248k	15.3k	1.1k	28.15	60.54	302.9k	1.23M	11.1k	10.7k
LightMem	50.18	61.36	37.9k	42.2k	2.2k	20	44.36	68.80	70.4k	19.2k	700	7	14.67	71.65	951.6k	351k	5.4k	35
MemoryOS	48.11	58.36	210.3k	131k	7.2k	11	37.80	67.80	532.5k	78.8k	8.2k	36	16.32	78.74	1.89M	713k	6.6k	354
A-MEM	42.27	54.03	3.64M	665k	16.7k	12	45.89	59.40	3.31M	357k	12.2k	10	25.62	57.95	2.73M	1.64M	3.6k	29
StructMem	53.31	67.40	197.8k	67.5k	6.4k	23	46.45	74.00	281.7k	55.7k	821	7	14.45	58.37	1.48M	386k	5.4k	35
FlashMem	**59.36**	**77.60**	46k	11.2k	1.1k	14	**50.49**	83.20	126.3k	35.5k	1.7k	88	**38.80**	**81.88**	299.8k	35.3k	1.1k	95
FlashMem†	54.03	75.45	46k	11.2k	496	15	50.15	**84.60**	126.3k	35.5k	817	98	36.61	80.64	299.8k	35.3k	514	109
FlashMem‡	53.17	72.46	15k	8.7k	493	16	48.37	81.20	39.1k	30.3k	815	99	35.66	80.73	91.2k	58.0k	516	112

Table 3: Evidence-view ablation on LoCoMo, LME-O, and LME-S with the Qwen3-8B inference model (accuracy).

Dataset	S	R	S+L	S+R
LoCoMo	51.3	45.2	53.1	**56.1**
LME-O	45.0	46.6	53.6	**54.6**
LME-S	44.4	44.0	57.3	**61.4**

Table 4: Selector comparison on LoCoMo with the GLM-4.5-Air inference model; ours uses Qwen2.5-0.5B (accuracy/F1).

ρ	CacheBlend	FusionRAG	Ours
0.1	51.7/43.4	53.2/42.1	**55.8/45.0**
0.2	53.8/45.9	58.2/48.5	**59.0/50.0**
0.3	58.2/47.5	60.7/49.0	**62.7/50.3**

Table 5: Qwen2.5-0.5B selector-layer aggregation on LME-O with the GLM-4.5-Air inference model (F1).

ρ	Last layer	Later half	All layers
0.1	51.9	53.2	53.4
0.2	51.6	54.3	54.2
0.3	52.0	55.0	55.1

Table 5 compares layer aggregation in the Qwen2.5-0.5B selector on LME-O, using GLM-4.5-Air for inference and memory recomputation ratios $\rho = 0.1/0.2/0.3$. Later-half aggregation achieves relative F1 gains of 2.5%/5.2%/5.8% over final-layer-only selection, respectively, while closely matching all-layer aggregation with half the selector-key storage.

6.3 Efficiency Analysis

RQ3: How does FlashMem reduce construction and inference overhead? Figure 7 compares six memory methods on LoCoMo/Qwen3-8B. SimpleMem/A-MEM/FlashMem save 29.3k/854k/35.4k construction prefill tokens and 11.34k/3.08k/0.601k answering prefill tokens per query (panels a,b). Baselines use 30% recomputation projections; FlashMem uses measured counts. Median TTFT and mean answer-latency speedups are 2.72/2.98/2.71× and 2.66/3.01/2.81×, respectively (panels c,d).

Figure 8 plots projected token-equivalent savings relative to raw RAG, which has no memory construction cost. Q is the query count, and Q^* marks the zero crossing where query savings offset construction cost; Appendix B.2 details the cost model. On LoCoMo10, FlashMem breaks even after approximately 258 queries, versus 1,324 and 4,724 for LightMem and StructMem. Our framework lowers StructMem's construction and inference costs, bringing its projected break-even point within this workload.

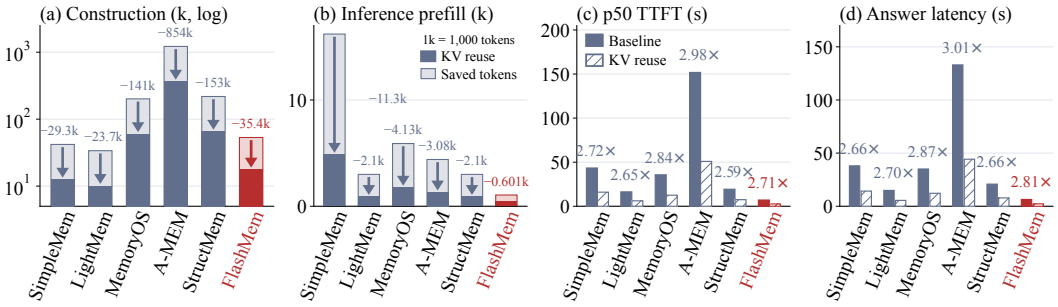

Figure 7: **Construction and inference with KV reuse on LoCoMo.** (a,b) Prefill savings: 30% recomputation projections for baselines, measured counts for FlashMem. (c,d) Baseline/reuse latency; labels show speedups.

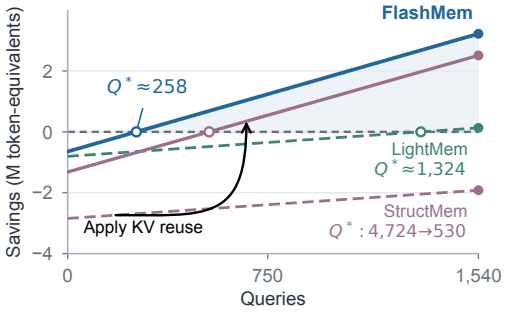

Figure 8: **Projected savings over repeated queries.** LoCoMo/Qwen3-8B versus raw RAG; Q^* marks token-equivalent break-even (Appendix B.2).

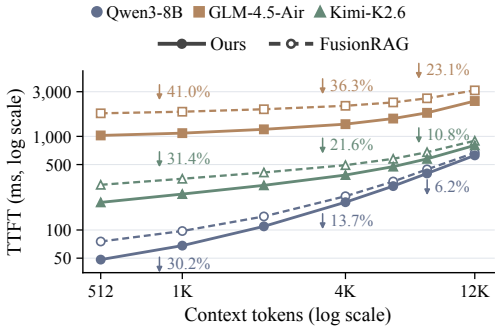

Figure 9: **TTFT across context lengths.** Paired means at 30% recomputation; labels show reductions versus FusionRAG.

RQ4: How does FlashMem reduce PIC execution and storage overhead? Figure 9 compares our method at 30% recomputation. With 1K cached tokens, profiled TTFT falls by 30.2%/41.0%/31.4% on Qwen3-8B/GLM-4.5-Air/Kimi-K2.6, respectively. Relative gains grow at shorter contexts, where selection accounts for more of PIC execution. Figure 10 shows 86.9%/89.6%/93.7% lower persistent cache capacity than full-history caching on LoCoMo/LME-S/HaluMem with Qwen3-8B. For FlashMem, the reported footprint includes both target-model KV states and the selector model's cached keys. The dashed raw-history references use the same target-model KV format. This reflects our co-design: summaries provide organized context, while complementary retrieval within their associated source regions supplies query-specific details. This keeps summaries compact for persistent KV reuse.

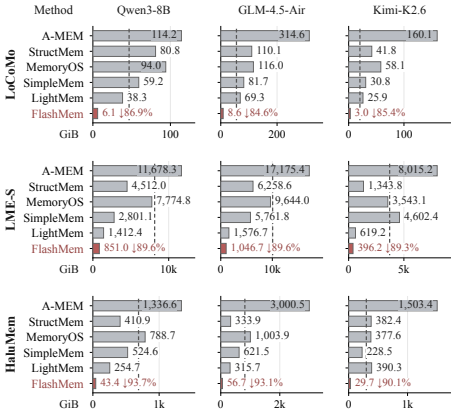

Figure 10: **Persistent KV storage.** BF16 target KV plus estimated selector keys for FlashMem (Appendix B.1). Dashed lines: raw history; red: reductions from raw history.

7 Conclusion

In this paper, we presented FlashMem, which co-designs compact memory with lightweight KV reuse. One-pass construction and source-linked retrieval preserve access to original evidence while reducing processing and persistent cache costs. Reuse across construction and inference reduces repeated prefill, while a small selector lowers the overhead of choosing recomputation positions.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Ding Chen, Simin Niu, Kehang Li, Peng Liu, Xiangping Zheng, Bo Tang, Xinchi Li, Feiyu Xiong, and Zhiyu Li. Halumem: Evaluating hallucinations in memory systems of agents. *arXiv preprint arXiv:2511.03506*, 2025.

Han Chen, Zining Zhang, Wenqi Pei, Bingsheng He, Ming Wu, Jason Zeng, Michael Heinrich, Wei Wu, and Hongbao Zhang. Memforest: An efficient agent memory system with hierarchical temporal indexing. *arXiv preprint arXiv:2605.23986*, 2026.

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

Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pp. 758–759, 2009.

Jizhan Fang, Xinle Deng, Haoming Xu, Ziyan Jiang, Yuqi Tang, Ziwen Xu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, et al. Lightmem: Lightweight and efficient memory-augmented generation. In *International Conference on Learning Representations*, volume 2026, pp. 98706–98729, 2026.

Yingsheng Geng, Yuchong Gao, Weihong Wu, Guyue Liu, and Jiang Liu. Relaycaching: Accelerating llm collaboration via decoding kv cache reuse. *arXiv preprint arXiv:2603.13289*, 2026.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. Epic: Efficient position-independent caching for serving large language models. *arXiv preprint arXiv:2410.15332*, 2024.

Zhanghao Hu, Qinglin Zhu, Runcong Zhao, Di Liang, Hanqi Yan, Yulan He, and Lin Gui. Beyond rag for agent memory: Retrieval by decoupling and aggregation. *arXiv preprint arXiv:2602.02007*, 2026.

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

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A human-inspired reading agent with gist memory of very long contexts. *arXiv preprint arXiv:2402.09727*, 2024.

Ruizhe Li, Licheng Zhang, Benfeng Xu, Mingxuan Du, Zheren Fu, and Weidong Chen. When your agent opens the chat app: Agent-controlled search over raw chat logs rivals structured memory. *arXiv preprint arXiv:2608.12888*, 2026.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 6342–6353, 2023.

Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Simplemem: Efficient lifelong memory for llm agents. *arXiv preprint arXiv:2601.02553*, 2026.

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

Yasmine Omri, Ziyu Gan, Zachary Broveak, Robin Geens, Zexue He, Alex Pentland, Marian Verhelst, Tsachy Weissman, and Thierry Tambe. Agent memory: Characterization and system implications of stateful long-horizon workloads. *arXiv preprint arXiv:2606.06448*, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 3982–3992, 2019.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and trends® in information retrieval*, 4(1-2):1–174, 2009.

Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Jiahao Wang, Weiyu Xie, Mingxing Zhang, Boxin Zhang, Jianwei Dong, Yuening Zhu, Chen Lin, Jingqi Tang, Yaochen Han, Zhiyuan Ai, et al. From prefix cache to fusion rag cache: Accelerating llm inference in retrieval-augmented generation. *Proceedings of the ACM on Management of Data*, 4(1 (SIGMOD):1–28, 2026a.

Kuan Wang. Memdelta: Controlled baselines and hidden confounds in agent memory evaluation. *arXiv preprint arXiv:2606.29914*, 2026.

Shihao Wang, Jiahao Chen, Yanqi Pan, Hao Huang, Yichen Hao, Xiangyu Zou, Wen Xia, Wentao Zhang, Chongyang Qiu, and Pengfei Wang. Prophetkv: User-query-driven selective recomputation for efficient kv cache reuse in retrieval-augmented generation. *arXiv preprint arXiv:2602.02579*, 2026b.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024.

Buqiang Xu, Yijun Chen, Jizhan Fang, Ruobin Zhong, Yunzhi Yao, Yuqi Zhu, Lun Du, and Shumin Deng. Structmem: Structured memory for long-horizon behavior in llms. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 122–146, 2026a.

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

Xiucheng Xu, Bingbing Xu, Tian Xueyun, Zihe Huang, Rongxin Chen, Li Yunfan, and Huawei Shen. Chain-of-memory: Lightweight memory construction with dynamic evolution for llm agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11618–11631, 2026c.

Jianxin Yan, Wangze Ni, Zhenxin Li, Jiabao Jin, Zhitao Shen, Haoyang Li, Jia Zhu, Peng Cheng, Xuemin Lin, Lei Chen, et al. Qcfuse: Query-aware cache fusion via compressed view for efficient rag serving. *arXiv preprint arXiv:2606.05875*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. Kvlink: Accelerating large language models via efficient kv cache reuse. *Advances in Neural Information Processing Systems*, 38: 133797–133824, 2026.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*, pp. 94–109, 2025.

Hancheng Ye, Zhengqi Gao, Mingyuan Ma, Qinsi Wang, Yuzhe Fu, Ming-Yu Chung, Yueqian Lin, Zhijian Liu, Jianyi Zhang, Danyang Zhuo, et al. Kvcomm: Online cross-context kv-cache communication for efficient llm-based multi-agent systems. *Advances in Neural Information Processing Systems*, 38:17882–17928, 2026.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Jiaquan Zhang, Chaoning Zhang, Shuxu Chen, Zhenzhen Huang, Pengcheng Zheng, Zhicheng Wang, Ping Guo, Fan Mo, Sung-Ho Bae, Jie Zou, et al. Lightweight llm agent memory with small language models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12914–12929, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024.

Yongjie Zhou, Shuai Wang, Bevan Koopman, and Guido Zuccon. Reproducing lightmem: Naive rag is just as good for memory management. *arXiv preprint arXiv:2607.29104*, 2026.

Appendix Contents

Page

A Experimental Setup and Implementation 14
A.1 Models and Server-Side Setup 14
A.2 Datasets 14
A.3 Baselines 14
A.4 Evaluation Protocol 15
A.5 Hyperparameters 15
A.6 Agent Memory Configuration 16

B Supplementary Evaluation Details 18
B.1 Persistent KV Storage Accounting 18
B.2 Break-even with Recomputation 18

C Metric Definitions for the Motivation 21
C.1 First-Gold Mean Reciprocal Rank 21
C.2 Token Surprisal and Its Concentration 21

D KV Reuse: Implementation and Cost 22
D.1 KV Reuse Procedure 22
D.2 RoPE Correction 25
D.3 Selection Across Different Tokenizers 26

A Experimental Setup and Implementation

We describe component-study settings, benchmark protocols, and baseline implementations. Maintable and diagnostic configurations are identified separately where they differ.

A.1 Models and Server-Side Setup

Models. Qwen3-8B (Yang et al., 2025) is the default model for both inference and construction for our component studies. It constructs summaries and answers questions from the retrieved evidence. The summary study in Figure 2 varies construction- and inference-model sizes. The main comparison table evaluates three inference backbones—Qwen3-8B, GLM-4.5-Air (Zeng et al., 2025), and Kimi-K2.6 (Team et al., 2025). The selector model is Qwen2.5-0.5B. The semantic judging protocol and default parameters appear in Appendices A.4 and A.5.

Server-side configuration. The available server-side hardware is listed in Table 6; the host runs Ubuntu 22.04. Benchmark generation and semantic judging use separate endpoints.

A.2 Datasets

LoCoMo. LoCoMo (Maharana et al., 2024) evaluates memory over 10 multi-session conversations. We use 1,540 non-adversarial questions covering single-hop, multi-hop, temporal, and open-domain reasoning. Individual diagnostic studies use subsets determined by the availability of valid evidence annotations or judgments; their sample sizes and exclusions are specified with their results. Token counts depend on the tokenizer and history serialization and are reported separately for each measurement.

LongMemEval. LongMemEval (Wu et al., 2024) evaluates information extraction, multi-session reasoning, temporal reasoning, knowledge updates, and abstention. The main comparison uses LongMemEval-S (LME-S). Component and supporting studies also use LongMemEval-O (LME-O), with the same 500 questions: *LongMemEval-O* retains the evidence sessions, whereas *LongMemEval-S* includes additional conversation history. Comparing the two measures performance with and without that additional retrieval and reading burden; it does not isolate memory representation quality. The raw-history analysis in Appendix C.2 uses the S variant.

HaluMem. HaluMem (Chen et al., 2025) evaluates hallucination resistance of memory systems at the operation level: memory extraction, memory update, and memory question answering. Each of 20 simulated users carries a persona-grounded history of roughly 70 sessions ($\sim$160K tokens per user, 3,467 questions in the Medium version we use), interleaving factual dialogues with distractor content; questions span fact recall, dynamic updates, multi-hop inference, and conflict resolution. For the main comparison (Table 2), each question is anchored to the session in which it is asked, and systems may retrieve only from that session's causal past, testing whether memory construction preserves correct facts under interleaved distractors and updates. The information-theoretic analysis (§C.2) additionally scores each user's full concatenated history as one unit.

A.3 Baselines

The baselines differ in their stored memory units and retrieval procedures. We retain those procedures in the reproduced configurations rather than replacing every system's retriever with the retrieval stack of FlashMem. The descriptions below distinguish benchmark evaluation from the earlier localization study, whose system coverage is narrower.

Raw-turn RAG. Raw RAG retrieves directly from benchmark turns without memory construction. Its main-table configuration is separate from the raw-only component setting in Table 3.

LightMem. LightMem (Fang et al., 2026) organizes memory through compression, short-term summarization, and long-term consolidation. In the reproduced configuration, turns are rewritten into self-contained *fact entries* that serve as both its first-layer memory and its dense-retrieval inference-time pool; every entry records its source turn.

A-MEM. A-MEM (Xu et al., 2026b) constructs notes containing a contextual description, keywords, and tags. Insertion links a note to related notes and may update their attributes. Inference-time retrieval searches the note contents and indexed attributes; our HaluMem harness uses MiniLM embeddings and retrieves the top 10 notes.

MemoryOS. MemoryOS (Kang et al., 2025) maintains short-term interactions, mid-term session memory, and long-term user and assistant information. Inference retrieves from these tiers into one inference prompt. The baseline retains its own update and retrieval procedures, including its MiniLM embedding configuration in the HaluMem reproduction.

StructMem. StructMem (Xu et al., 2026a) organizes events and their relationships in a hierarchy. The reproduced configuration builds first-layer *event items* and second-stage *session summaries*, each covering the items of one session; inference retrieves both through independent dense retrievals into a single inference prompt. The localization-study configuration retrieves 60 entries and five summaries.

SimpleMem. SimpleMem (Liu et al., 2026) constructs *atomic facts* with resolved references and timestamps, merges related context, and plans retrieval for each query. It is included in the benchmark comparisons; our HaluMem harness wraps the released implementation to count construction and inference model usage, using its configured Qwen3-Embedding-0.6B embedding endpoint. The localization study evaluates LightMem and StructMem memory pools only.

A.4 Evaluation Protocol

Metrics. We report normalized token-level F1 and semantic accuracy judged against the reference answer. Logical token usage counts model input and output separately for construction and inference; inference usage covers model calls from evidence retrieval through answer generation. Construction prefill counts fresh and recomputed target positions separately from logical input tokens.

Judge. Accuracy is judged by GLM-5.3 under a single-field JSON constraint, using the decoding settings in Table 6. The judge is method-blind: it sees only the question, reference, and candidate answer. The evaluation uses the prompt below. Historical component studies identify their paired judgment batches and sampling units alongside their results; absolute scores from separate runs are not treated as interchangeable. The system prompt:

```
You are a strict, method-blind evaluator of question answering. Judge
only whether the candidate answer is semantically correct according to
the question and reference answer. Do not infer which system produced
it.
```

The user template instantiates the question, reference, and candidate under the following rules:

```
Decide whether the candidate answer is correct.
Rules: 1. Accept concise paraphrases, equivalent names, equivalent
date/number formats, and a correct answer embedded in harmless extra
explanation. 2. Reject a wrong person, entity, event, date,
ordering, count, amount, or polarity; a contradiction; a refusal when
the reference answers the question; or an answer missing a required
list item, comparison, calculation, or event. 3. Extra text is
harmless only if it does not add a materially false answer claim. 4.
For open-ended preference or recommendation questions, the answer need
not copy every example in the reference, but it must correctly use the
core personal information required by the reference. 5. Treat the
reference as the scoring ground truth. Do not use outside knowledge.
Return exactly one JSON object with one boolean field and no other
text: {"correct": true} or {"correct": false}
```

A.5 Hyperparameters

Table 6 lists the component-study settings for FlashMem and the semantic judge. It does not impose the same decoding or retrieval configuration on every baseline. In particular, BGE-M3, hybrid

retrieval, and cross-encoder reranking describe our pipeline; the localization study uses MiniLM, and the HaluMem baseline harnesses retain their configured embedding models. Historical component studies specify their own budgets and controls alongside their results. Our serving stack is developed on SGLang (Zheng et al., 2024). The recorded baseline settings include the following: A-MEM retrieves the top $k = 10$ notes per query; MemoryOS caps short-term memory at 5 turns and mid-term memory at 2,000 entries; LightMem compresses with rate $r = 0.6$ and admits turns into its short-term buffer at a threshold of $\mathrm{th} = 512$ tokens.

Table 6: Component-study configuration of FlashMem and the semantic judge. Main-table settings and individual diagnostic studies may differ.

Category	Setting
Default decoding (inference / construction / judge)	
Temperature	0
Top-p	1
Random seed	42
Inference max output tokens	512
Construction max output tokens	16,000
Judge max output tokens	8,192
Thinking mode	Disabled
Retrieval (ours)	
Embedding model	BGE-M3
Cross-encoder	`bge-reranker-v2-m3`
Summary candidates k_{m}	40
Construction (ours)	
Segment size	$\leq$48 sources or 10,000 source tokens
Source grouping	Adjacent canonical units in pairs
Server-side execution	
Hardware	8$\times$ NVIDIA H20 (96 GB)
Host memory	1.9 TB
CPU	2$\times$ Intel Xeon Platinum 8457C

A.6 Agent Memory Configuration

Construction. Within each session, adjacent canonical source units are grouped in pairs and segmented under the limits in Table 6. The construction model returns item bodies, source identifiers, and omission records. The parser validates identifiers, removes omission labels for cited sources, and records otherwise unaccounted sources. Deterministic postprocessing attaches source metadata and extracts exact-value strings, including numbers, units, and URLs, from each item body.

Memory reading. The Read operation in Equation 1 retrieves and assembles evidence as specified in Equation 4. It first uses hybrid retrieval and cross-encoder reranking to select complete memory summaries under target budget β_{m}. The source identifiers of the summaries retained in S_q define a deduplicated pool $\mathcal{H}_q$ of original source units. The same query then retrieves and reranks up to k_{h} raw candidates within this pool; an empty pool yields no raw candidates. Summary rendering A_{m} takes the selected set S_q; raw-window assembly A_{h} takes the query, raw candidates, and budget β_{h}. Algorithm 1 specifies both whole-item summary selection and raw-window assembly. Budgets count the complete rendered evidence, including timestamps and speaker labels. The highest-ranked summary is retained in full if it alone exceeds the cap; subsequent non-fitting items are skipped. Raw windows are verbatim contiguous source spans, admitted only when the complete raw section fits its cap. Relevance ties follow source and span order; each view is rendered chronologically.

Inference. The evidence E_q returned by Read is passed to the inference model, implementing $a = F_\theta(q, E_q)$ in Equation 1. The prompt separates summaries, original evidence, question time, and the question, followed by an explicit answer delimiter. One generation produces the answer. Selector-guided KV selection adds the forward pass described in Section 5.2.

Algorithm 1 FlashMem: Memory Construction, Reading, and Answer Generation

Input: Construction model F_{c}, inference model F_θ, instructions $\pi_{\mathrm{c}}, \pi_{\mathrm{i}}$; candidate limits $k_{\mathrm{m}}, k_{\mathrm{h}}$; token budgets $\beta_{\mathrm{m}}, \beta_{\mathrm{h}}$
Output: Updated summaries $\mathcal{M}$ on construction; evidence E_q from READ; answer a from INFER

function CONSTRUCTMEMORY($\mathcal{H}_{\mathrm{new}}, \mathcal{M}$)
  for each chronological segment B_j of $\mathcal{H}_{\mathrm{new}}$ **do**
    $\Delta\mathcal{M}_j \leftarrow F_{\mathrm{c}}(\pi_{\mathrm{c}} \| B_j)$ ▷ One construction-model generation
    Validate source identifiers and attach metadata to $\Delta\mathcal{M}_j$
    $\mathcal{M} \leftarrow \mathcal{M} \uplus \Delta\mathcal{M}_j$ ▷ Append complete items
  end for
  return $\mathcal{M}$ ▷ Keep the original history as text
end function
function READ($q, \mathcal{D}_t; \beta$)
  $(\mathcal{M}_t, \mathcal{H}_t) \leftarrow \mathcal{D}_t$; $(\beta_{\mathrm{m}}, \beta_{\mathrm{h}}) \leftarrow \beta$
  $D_{\mathrm{m}} \leftarrow$ RETRIEVEANDRERANK($q, \mathcal{M}_t, k_{\mathrm{m}}$)
  $S_q \leftarrow \emptyset$
  for each item m in D_{m}, in decreasing relevance order **do**
    if $|A_{\mathrm{m}}(S_q \cup \{m\})| \le \beta_{\mathrm{m}}$ **or** m is the first item **then**
      $S_q \leftarrow S_q \cup \{m\}$ ▷ Retain whole items; skip other non-fitting items
    end if
  end for
  $\mathcal{H}_q \leftarrow \bigcup_{m \in S_q} \mathrm{Src}(m)$ ▷ Source units cited by selected summaries
  $D_{\mathrm{h}} \leftarrow$ RETRIEVEANDRERANK($q, \mathcal{H}_q, k_{\mathrm{h}}$)
  return $E_q \leftarrow A_{\mathrm{m}}(S_q) \| \textsc{AssembleRaw}(q, D_{\mathrm{h}}; \beta_{\mathrm{h}})$ ▷ Equation 4
end function
function ASSEMBLERAW($q, D_{\mathrm{h}}; \beta_{\mathrm{h}}$) ▷ Implements A_{h}
  $R_q \leftarrow \emptyset$
  Score each span v_{ij} in each candidate source $i \in D_{\mathrm{h}}$ by $r(q, v_{ij})$
  $j_i^\star \leftarrow \arg\max_j r(q, v_{ij})$ and $p_i \leftarrow r(q, v_{ij_i^\star})$ for each source i
  for each source i in decreasing p_i **do**
    if $|\,\mathrm{Render}(R_q \cup \{v_{ij_i^\star}\})| \le \beta_{\mathrm{h}}$ **then**
      $R_q \leftarrow R_q \cup \{v_{ij_i^\star}\}$ ▷ Initialize $W_i = [j_i^\star, j_i^\star]$
    end if
  end for
  repeat
    $R_{\mathrm{old}} \leftarrow R_q$
    for each source i represented in R_q, in the same priority order **do**
      $R_q \leftarrow$ EXTENDIFFITS($R_q, i, q, \beta_{\mathrm{h}}$)
    end for
  until $R_q = R_{\mathrm{old}}$
  return $\mathrm{Render}(R_q)$ ▷ Chronologically ordered raw windows
end function
function INFER($q, \mathcal{D}_t$)
  $E_q \leftarrow$ READ($q, \mathcal{D}_t; (\beta_{\mathrm{m}}, \beta_{\mathrm{h}})$)
  $X_q \leftarrow \pi_{\mathrm{i}} \| E_q \| q$
  return $a \leftarrow F_\theta(X_q)$ ▷ One generation; KV execution in Algorithm 2
end function

Complete procedure. Algorithm 1 separates CONSTRUCTMEMORY, READ, and INFER. Construction updates the summary store once per new segment while retaining the original history as text, yielding the state $\mathcal{D}_t = (\mathcal{M}_t, \mathcal{H}_t)$ at query time. READ returns E_q; INFER uses it in one answer-generation call. The helper RETRIEVEANDRERANK implements R through hybrid search and cross-encoder reranking over the supplied pool: $\mathcal{M}_t$ for summaries, then $\mathcal{H}_q$ for source evidence. Here, $\mathrm{Src}(m)$ resolves the source identifiers recorded during construction to original units in $\mathcal{H}_t$; set union removes duplicates. The operator A_{m} renders selected summaries, and ASSEMBLERAW implements A_{h}, returning rendered raw windows. Both A_{m} and Render order their inputs chronologically and include source metadata; all budget checks count these fully rendered inference-model tokens. The prompt X_q makes explicit the fixed instructions and serialization suppressed in $F_\theta(q, E_q)$. For each represented source, EXTENDIFFITS tests the two adjacent boundary spans in decreasing reranker score and accepts at most one fitting extension per round; ties follow source and span order.

B SUPPLEMENTARY EVALUATION DETAILS

This appendix records the settings behind the brief component comparisons, the persistent-cache accounting, and the execution-cost analysis in Section 6.

B.1 PERSISTENT KV STORAGE ACCOUNTING

Figure 10 converts memory-pool token counts into persistent cache capacity for each inference model, including target KV and selector keys for FlashMem. Raw-history and memory counts use the corresponding inference-model tokenizer. MemoryOS counts one page per source conversation record to avoid duplicating the same record across storage tiers. Partial pools are completed using the mean tokens per completed sample and the full sample count. Missing pools use the mean memory/raw ratio from available pools for the same dataset and method, falling back to the same model and method.

For a pool containing N_θ target-model tokens, target KV occupies $N_\theta b_\theta$ bytes. Under BF16, b_θ is 147,456 for Qwen3-8B, 188,416 for GLM-4.5-Air, and 70,272 for Kimi-K2.6 with compressed MLA. FlashMem additionally retains the selector's keys at layers $\mathcal{L}$ for all cached selector tokens, not only the positions selected for recomputation. Its total persistent capacity is

$$C_{\text{FlashMem}} = N_\theta b_\theta + N_\phi b_\phi, \qquad b_\phi = 2|\mathcal{L}|H_{\phi,\text{KV}} d_\phi,$$

where N_ϕ uses the selector tokenizer, $H_{\phi,\text{KV}}$ is its number of KV heads, and the factor of two is the BF16 bytes per element. Only keys are retained for the selector. For Qwen2.5-0.5B with the later 12 layers, two KV heads, and head dimension 64, $b_\phi = 3{,}072$ bytes per selector token.

Figure 10 estimates the selector term using $N_\phi \approx N_\theta$, giving $C_{\text{FlashMem}} \approx N_\theta(b_\theta + 3{,}072)$. This equal-token-count approximation yields the extra storage ratios in Table 1: 2.08%, 1.63%, and 4.37% for the three targets, respectively; actual overhead scales with N_ϕ/N_θ. The table isolates this component, which is also included in Figure 10. Raw-history caching and the other memory methods use target KV only. For N_{raw} raw-history tokens, the FlashMem reduction is therefore $1 - C_{\text{FlashMem}}/(N_{\text{raw}} b_\theta)$, rather than the memory/raw token reduction alone. Capacities are reported in GiB (2^{30} bytes); raw text and retrieval indices remain stored separately.

B.2 BREAK-EVEN WITH RECOMPUTATION

Cumulative cost and break-even query count. Consider a fixed collection of conversations whose memory is constructed once and reused for subsequent queries. Let $Q \in \mathbb{Z}_{\geq 0}$ be the total number of queries served over this collection, $B_m \geq 0$ the one-time construction cost of method m, and $c_{m,q} \geq 0$ its cost of inference for query q. The cumulative cost is

$$C_m(Q) = B_m + \sum_{q=1}^{Q} c_{m,q}. \tag{6}$$

For the projection, we hold the mean per-inference cost c_m fixed as Q varies, giving $C_m(Q) = B_m + Qc_m$. Raw RAG is the reference method, with no memory-construction cost, $B_0 = 0$, and mean inference cost $c_0 > 0$, so $C_0(Q) = Qc_0$. The cumulative savings relative to raw RAG are

$$S_m(Q) = C_0(Q) - C_m(Q) = Q(c_0 - c_m) - B_m. \tag{7}$$

Thus construction is amortized when recurring query savings cover its initial cost, i.e., $S_m(Q) \geq 0$. For $B_m > 0$, define the break-even threshold on the continuous cost curves by

$$Q_m^* := \inf\{x \geq 0 : B_m + xc_m \leq xc_0\} = \begin{cases} \dfrac{B_m}{c_0 - c_m}, & c_m < c_0, \\ +\infty, & c_m \geq c_0, \end{cases} \tag{8}$$

where $\inf \emptyset = +\infty$. A finite Q_m^* is the intersection of the two curves; method m is strictly cheaper for $Q > Q_m^*$. When Q_m^* is finite, the smallest integer query count with $C_m(Q) \leq C_0(Q)$ is $\lceil Q_m^* \rceil$, and the first strictly cheaper count is $\lfloor Q_m^* \rfloor + 1$. The figures report the continuous threshold, rounded to the nearest query. For $Q > 0$, the normalized cumulative cost is

$$r_m(Q) = \frac{C_m(Q)}{C_0(Q)} = \frac{B_m}{Qc_0} + \frac{c_m}{c_0}, \tag{9}$$

with $r_m(Q) \leq 1$ indicating that construction has been amortized.

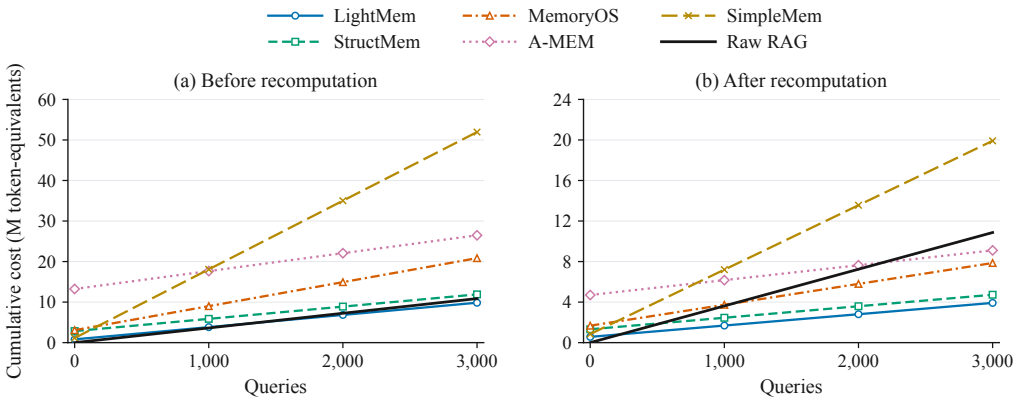

Figure 11: **Cumulative cost before and after recomputation.** LoCoMo, Qwen3-8B, under the conditional cost model described here. The panels share the query range but use different vertical scales. Colors identify methods consistently across panels. The black line is raw RAG; each intersection marks Q_m^*, beyond which the method costs less. Query counts beyond the 1,540-question workload extrapolate constant mean inference cost.

Cost units and workload. We instantiate this model using the rounded base Qwen3-8B/LoCoMo entries in Table 2. Costs are token-equivalent units, with executed prefill and output tokens weighted equally. Let b_m be the sum of the reported construction input and output counts. Under the assumption that these counts are per-conversation means, we use $B_m = 10b_m$ for the 10-conversation collection. Inference input and output counts determine c_m, with $c_0 = 3{,}622$ for raw RAG. The evaluated workload contains $Q_{\text{eval}} = 1{,}540$ queries in total; a method amortizes construction within this workload if $Q_m^* \leq Q_{\text{eval}}$. The collection is held fixed as Q increases, with no further memory construction or updates included in this projection.

Applying our recomputation to other memory pipelines. Figure 11 compares five external pipelines using the construction-prefill projections in Figure 7(a), with 30% recomputation for these pipelines. Let P_m and $P'_m = 0.3P_m$ denote construction-prefill token counts, treated as per-conversation quantities, before and after recomputation, and t_m and t'_m the reported mean per-round answer-generation times. We compare the baseline with construction reuse alone and with a combined construction-and-inference projection, assuming the execution measurements apply to the same workload as the base costs:

$$C_m^{\text{constr}}(Q) = \widetilde{B}_m + Qc_m, \qquad \widetilde{C}_m(Q) = \widetilde{B}_m + Q\widetilde{c}_m,$$

where $\widetilde{B}_m = B_m - 10(P_m - P'_m)$ and $\widetilde{c}_m = c_m(t'_m/t_m)$. The construction-only case leaves inference cost unchanged; the combined case assumes that the answer-generation latency ratio transfers to the token-equivalent inference-cost proxy. Construction output is unchanged, and common-prefix hits are not counted again.

Break-even outcomes. Under the combined projection, the number of methods with finite crossings increases from two to four. MemoryOS and A-MEM newly become amortizable; LightMem and StructMem already have finite crossings and reach them earlier. Within the 1,540-query workload, the number breaking even rises from one to three: LightMem, StructMem, and MemoryOS, with normalized costs of 0.41, 0.55, and 0.87. A-MEM needs more queries, remaining at 1.25 within the workload. SimpleMem still has no finite crossing because its projected inference cost is about 6,357 units, above raw RAG's 3,622.

The role of recurring query savings. Construction reuse alone reduces the LightMem and StructMem crossings to approximately 935 and 2,191 queries; only LightMem falls below raw RAG at 1,540 queries. It cannot create a finite crossing when a method's inference cost remains higher than raw RAG's. The base FlashMem configuration already crosses at about 258 queries and costs 0.42× raw RAG at this workload; the before/after comparison here is restricted to the five external pipelines with construction-prefill projections.

Scope of the projection. The recomputation curves are conditional cost projections. Scaling query cost by the mean answer-generation latency ratio assumes proportional changes in these two quantities; it does not measure a reduction in logical inference tokens or monetary cost. The projection uses the reported construction totals, assumes the per-conversation aggregation and workload correspondence specified above, omits additional cache setup cost, and keeps raw RAG unaccelerated. Query counts beyond Q_{eval} extrapolate the same mean inference costs. The break-even thresholds therefore describe this cost model for a fixed collection rather than measured end-to-end lifecycle savings.

C METRIC DEFINITIONS FOR THE MOTIVATION

C.1 FIRST-GOLD MEAN RECIPROCAL RANK

Retrieval units and gold evidence. Figure 4 ranks raw turns and LightMem fact entries in separate pools within each LoCoMo conversation. A raw turn is *gold* if annotated as evidence for the question; an entry is gold if its recorded source turn is annotated. Multiple entries can share a gold source. This criterion measures source-linked localization, without guaranteeing that an entry retains the answer-relevant details.

Ranking. BM25 uses lexical matching with $k_1 = 2.2$, $b = 0.75$, and a fixed stopword list (Robertson & Zaragoza, 2009). Dense retrieval uses cosine similarity between question and unit embeddings from the same MiniLM sentence encoder (Reimers & Gurevych, 2019). Hybrid retrieval uses reciprocal rank fusion (Cormack et al., 2009), assigning each unit score $1/(60 + r_{\text{dense}}) + 1/(60 + r_{\text{BM25}})$. Here, the two ranks are one-based; all rankings use fixed tie-breaking.

Metric and aggregation. For question q, let r_q be the one-based rank of the first gold unit in the retrieved list. First-gold mean reciprocal rank (MRR) is

$$\text{MRR}(\mathcal{Q}) = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \text{RR}(q), \qquad \text{RR}(q) = \begin{cases} 1/r_q, & \text{if a gold unit is retrieved,} \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

The set $\mathcal{Q}$ contains the questions for a representation, retriever, and distance group; higher MRR indicates earlier gold retrieval. Evidence distance is the session-index gap from the final session to the earliest annotated gold evidence, grouped into 1–5, 6–10, 11–15, 16–20, 21–25, and 26–31 sessions. The raw-turn and fact-entry cohorts contain 1,533 and 1,531 category 1–4 questions. Group membership, pool size, and gold multiplicity can differ, so the comparisons are descriptive.

C.2 TOKEN SURPRISAL AND ITS CONCENTRATION

History units and scoring context. Figure 3 uses 10 LoCoMo conversations, 500 LongMemEval-S question-specific haystacks, and 20 HaluMem-Medium user histories. Turns are serialized as `{speaker}: {text}` and joined by newlines, without chat templates, padding, or control tokens. Frozen Qwen3-8B in `bf16` scores each tokenized history $H = (x_1, \ldots, x_N)$ with teacher forcing and no generation. Each chunk contains up to 2,048 new tokens plus the preceding chunk's last token when available. Model state is otherwise reset, so each token sees only its chunk prefix, not the entire preceding history.

Token surprisal. Following the token-level self-information measure of Li et al. (2023), the surprisal of observed token x_i is

$$s_i = -\log_2 p_\theta(x_i \mid c_i), \tag{11}$$

where p_θ is the next-token distribution and c_i the visible prefix, including the carried token. Less predictable tokens have higher surprisal, measured in bits. Without an inserted beginning-of-sequence token, the first history token is unscored; every later token is scored once. All fractions and sums use the scored set $\mathcal{I}$, with $N_s = |\mathcal{I}|$.

Concentration curve and 90% coverage. Sort a history's scored values in descending order, $s_{(1)} \geq \cdots \geq s_{(N_s)}$. For positive total surprisal and $r \in [0, 1]$, define

$$C(r) = \frac{\sum_{j=1}^{\lceil rN_s \rceil} s_{(j)}}{\sum_{i \in \mathcal{I}} s_i}, \qquad C(0) = 0. \tag{12}$$

Thus, $C(r)$ is the surprisal share carried by the highest-scoring fraction r of tokens, using scores from their original contexts. An individual history's 90% coverage is the smallest k/N_s with $C(k/N_s) \geq 0.9$. The figure plots the unweighted mean of $C(r)$ across histories on a uniform 100-point grid, with a 25th–75th percentile band. Its legend's r_{90} is the 0.9 crossing of the *mean curve*, linearly interpolated between grid points, rather than the mean of per-history thresholds. These measures describe model-dependent predictability; they do not guarantee evidence preservation after token removal.

D KV REUSE: IMPLEMENTATION AND COST

This appendix details selector scoring, top-k selection, cross-tokenizer alignment, and the resulting selective KV updates.

D.1 KV REUSE PROCEDURE

Algorithm 2 gives the shared procedure for construction and inference calls. As in Section 5.2, I denotes selected target positions and $\mathcal{R}$ their eligible candidate set. For execution, $\mathcal{U}$ contains mandatory or uncached positions, $\mathcal{P}$ contains directly reusable positions, and $\mathcal{S} = I \cup \mathcal{U}$ contains all positions processed by the target. Persistent target-cache entries are denoted by $\mathcal{E}$. Loading the entries used by one call forms the assembled call-local cache KV^{cached}; selective updates produce KV^{updated}. The target has L_θ layers. Each target cache entry records a segment g's per-layer KV rows and original preceding text $\gamma_0(g)$; $\gamma_X(g)$ is its preceding text in the current prompt. For summary entries, $\gamma_0(g)$ is the fixed inference instruction. Compatible entries must match the model checkpoint, tokenizer, positional encoding, and cache format. Mandatory inputs and cache misses enter $\mathcal{U}$; inference calls include raw evidence, assembly metadata, and the query suffix in this set. Among other positions, matching preceding text permits direct reuse in $\mathcal{P}$; changed text produces candidates $\mathcal{R}$. A match is exact only for states originally obtained by full prefill; entries inherited from approximate execution remain approximate.

Selector-key preparation and scoring. During the same offline stage that builds a chunk's target KV cache, we encode its text with F_ϕ and retain selector keys only at layers $\mathcal{L}$. These keys are stored alongside the corresponding target entry as a separate model-specific cache. A missing selector-key entry must be prepared before that chunk can be scored; the cache-hit cost analysis assumes this preparation has already occurred. Online, the selector processes only task text τ. For inference, τ is the user question; for construction, it is a single-sentence question expressing the task objective. This question guides selection, while the target prompt X retains the full construction instructions. For scoring, the assembled chunk keys and task queries use a consistent position convention, with the task located after the scored chunks. The corresponding RoPE rotation is adjusted at each scoring layer; this aligns coordinates without adding cross-chunk context to the cached representations. Let m_{d} and n_{d} be the numbers of task tokens and assembled cached selector tokens, respectively. The selector has H_ϕ query heads of query–key dimension d_ϕ. At layer ℓ, let $Q^{(\ell)}_{\phi,h} \in \mathbb{R}^{m_{\mathrm{d}} \times d_\phi}$ contain the task queries and $K^{(\ell)}_{\phi,h} \in \mathbb{R}^{n_{\mathrm{d}} \times d_\phi}$ contain the aligned cached keys associated with query head h. For grouped-query attention, this association uses the corresponding KV head without materializing repeated keys. We expand Equation 5 into attention weights and token scores:

$$
\begin{aligned}
A^{(\ell)}_{\phi,h} &= \mathrm{softmax}_{\mathrm{row}}\left(\frac{Q^{(\ell)}_{\phi,h}(K^{(\ell)}_{\phi,h})^\top}{\sqrt{d_\phi}}\right), \\
s_\phi(j) &= \frac{1}{|\mathcal{L}| H_\phi m_{\mathrm{d}}} \sum_{\ell \in \mathcal{L}} \sum_{h=1}^{H_\phi} \sum_{u=1}^{m_{\mathrm{d}}} A^{(\ell)}_{\phi,h}[u, j], \qquad j = 1, \ldots, n_{\mathrm{d}}.
\end{aligned}
\tag{13}
$$

Each row is normalized over all assembled cached keys, before eligibility filtering. These products are used only for scoring: cached document keys and values do not participate in the selector's online hidden-state propagation.

Selection and tokenization. ALIGNTOSELECTOR maps eligible target positions $\mathcal{R}$ to cached selector-token positions $\mathcal{R}_{\mathrm{d}}$ through source-text offsets. We set $k = \lfloor \rho |\mathcal{R}_{\mathrm{d}}| \rfloor$ and select the k highest-scoring candidates to obtain $I_{\mathrm{d}} \subseteq \mathcal{R}_{\mathrm{d}}$. ALIGNTOTARGET maps their text intervals to overlapping eligible target tokens, yielding $I \subseteq \mathcal{R}$ (Section D.3). Mandatory positions do not consume this budget. Different token boundaries can change the realized target-token fraction, reported as $|I|/|\mathcal{R}|$ for nonempty $\mathcal{R}$. The selected set is shared across target layers.

Positional alignment and cache updates. For rotary positional embeddings, a cached key $k^{(\ell)}_0$ originally encoded at position p_0 is moved to position p by

$$
\widetilde{k}^{(\ell)}_p = R(p) R(p_0)^{-1} k^{(\ell)}_0, \qquad \widetilde{v}^{(\ell)}_p = v^{(\ell)}_0, \tag{14}
$$

where $R(p)$ is the rotary transformation applied per head. Appendix D.2 gives its derivation and conditions. Under the same RoPE convention, this changes the stored key's explicit positional rotation; it does not recover interactions with the new preceding context. The target initializes active positions $\mathcal{S} = I \cup \mathcal{U}$ from its own embeddings and updates them at each layer, with causal attention over the assembled cache. Selected positions still attend to unrecomputed cached states, so this execution remains approximate. Updates affect only the call-local cache; persistent entries remain unchanged. Construction and inference calls use the same selector selection, alignment, and sparse-update operations with their respective target model and prompt.

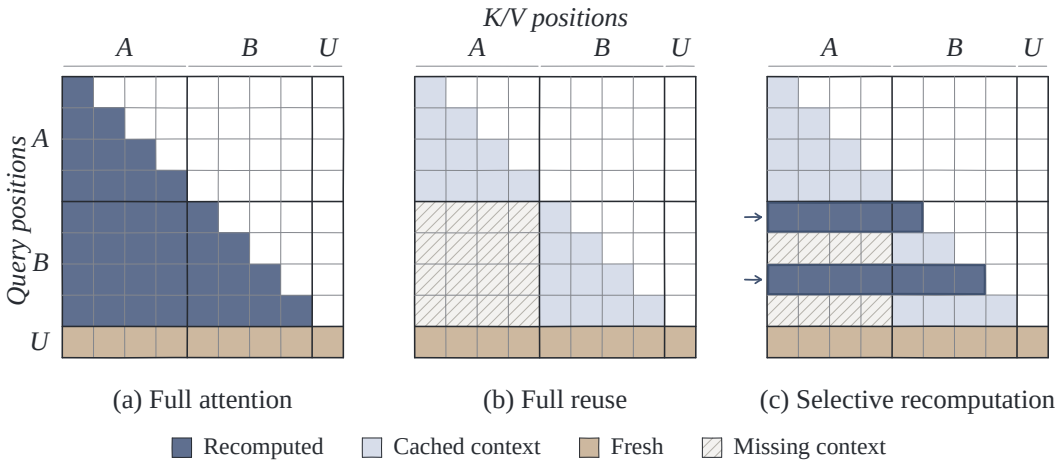

Figure 12: Attention with independently cached segments A and B and fresh input U. Direct reuse omits interactions from B to A (hatching); selective recomputation reevaluates selected query rows over their causal context. Fresh input U is processed in full in every setting. Cached blocks depict their original encoding context.

Algorithm 2 FlashMem: KV Reuse with a Lightweight Selector for Construction and Inference Calls

Input: Assembled prompt X; task text τ; mandatory-input segments; target F_θ, selector F_ϕ; target entries $\mathcal{E}$; selector-key entries $\mathcal{E}_\phi$; ratio $\rho \in [0, 1]$

Output: Generated output y (answer a for inference) and call-local target cache KV^{updated}

$\mathcal{P} \leftarrow \emptyset; \mathcal{R} \leftarrow \emptyset; \mathcal{U} \leftarrow \emptyset$
for each segment g of X, occupying target positions J_g **do**
  if g is a mandatory input or no compatible entry $\mathcal{E}[g]$ exists **then**
    $\mathcal{U} \leftarrow \mathcal{U} \cup J_g$ ▷ Mandatory inputs and cache misses
  else if $\gamma_0(g) = \gamma_X(g)$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup J_g$ ▷ Exact context match
  else
    $\mathcal{R} \leftarrow \mathcal{R} \cup J_g$ ▷ Unchanged text, changed context
  end if
end for
$I \leftarrow \emptyset$
if $\mathcal{R} \neq \emptyset$ **then**
  Prepare missing chunk keys in $\mathcal{E}_\phi$ at layers $\mathcal{L}$
  $\mathcal{R}_{\text{d}} \leftarrow \textsc{AlignToSelector}(X, \mathcal{R}, \mathcal{E}_\phi)$
  $K_\phi \leftarrow \textsc{AssembleSelectorKeys}(X, \mathcal{E}_\phi, \mathcal{L})$
  $Q_\phi \leftarrow \textsc{QueryPrefill}(F_\phi, \tau, \mathcal{L})$ ▷ Encode task text only
  Align Q_ϕ, K_ϕ to the scoring positions
  $s_\phi \leftarrow \textsc{ScoreCachedKeys}(Q_\phi, K_\phi)$ ▷ Equations (5) and (13)
  $k \leftarrow \lfloor \rho |\mathcal{R}_{\text{d}}| \rfloor$
  $I_{\text{d}} \leftarrow \textsc{TopK}(s_\phi, \mathcal{R}_{\text{d}}, k)$
  $I \leftarrow \textsc{AlignToTarget}(I_{\text{d}}, \mathcal{E}_\phi, X) \cap \mathcal{R}$
end if
$\mathcal{S} \leftarrow I \cup \mathcal{U}$ ▷ Mandatory inputs include the final prompt position
$KV^{\text{cached}} \leftarrow \textsc{LoadTargetRows}(\mathcal{E}, \mathcal{P} \cup \mathcal{R})$
$\{(\widetilde{K}_\theta^{(\ell)}, \widetilde{V}_\theta^{(\ell)})\}_{\ell=1}^{L_\theta} \leftarrow \textsc{AlignToPrompt}(KV^{\text{cached}}, X)$ ▷ Equation (14); values unchanged
$H[\mathcal{S}] \leftarrow \text{Embed}_\theta(X[\mathcal{S}])$
for $\ell = 1, \ldots, L_\theta$ **do**
  $(K_{\theta,\text{new}}^{(\ell)}, V_{\theta,\text{new}}^{(\ell)}) \leftarrow \textsc{TargetKV}(\ell, H[\mathcal{S}], \mathcal{S})$
  $(\widehat{K}_\theta^{(\ell)}, \widehat{V}_\theta^{(\ell)}) \leftarrow (\widetilde{K}_\theta^{(\ell)}, \widetilde{V}_\theta^{(\ell)})$ ▷ Aligned cached rows
  $(\widehat{K}_\theta^{(\ell)}[\mathcal{S}], \widehat{V}_\theta^{(\ell)}[\mathcal{S}]) \leftarrow (K_{\theta,\text{new}}^{(\ell)}, V_{\theta,\text{new}}^{(\ell)})$
  $H[\mathcal{S}] \leftarrow \textsc{TargetLayer}(\ell, H[\mathcal{S}], \widehat{K}_\theta^{(\ell)}, \widehat{V}_\theta^{(\ell)}, \mathcal{S})$ ▷ Causal attention
end for
$KV^{\text{updated}} \leftarrow \{(\widehat{K}_\theta^{(\ell)}, \widehat{V}_\theta^{(\ell)})\}_{\ell=1}^{L_\theta}$
$y \leftarrow \textsc{Decode}(F_\theta, H[|X|], KV^{\text{updated}})$
return (y, KV^{updated}) ▷ Persistent entries remain unchanged

D.2 RoPE Correction

When independently cached chunks are assembled into a new prompt, their keys must use the positions assigned by the assembled sequence. We first show that successive rotary positional transformations compose by adding their position offsets, then derive the correction and illustrate it with a two-chunk example.

Definition. Let $\bar{k} \in \mathbb{R}^d$ denote the rotary coordinates of an unrotated key, where d is even. We write adjacent coordinate pairs for clarity; a split-half implementation is equivalent up to a fixed permutation. For a fixed frequency schedule, define

$$\theta_i = b^{-2(i-1)/d}, \quad i = 1, \ldots, d/2, \qquad \boldsymbol{\theta} = (\theta_1, \theta_1, \ldots, \theta_{d/2}, \theta_{d/2}), \tag{15}$$

where b is the model's RoPE base (conventionally 10,000). Define the pairwise rotation operator

$$J\bar{k} = (-\bar{k}_2, \bar{k}_1, -\bar{k}_4, \bar{k}_3, \ldots, -\bar{k}_d, \bar{k}_{d-1}), \qquad J^2\bar{k} = -\bar{k}. \tag{16}$$

The rotary encoding at position s is

$$\mathrm{RoPE}(\bar{k}, s) = \bar{k} \odot \cos(s\boldsymbol{\theta}) + (J\bar{k}) \odot \sin(s\boldsymbol{\theta}) = R(s)\bar{k}, \tag{17}$$

where $\odot$ denotes elementwise multiplication and $R(s)$ is the corresponding block-diagonal rotation matrix. Sine and cosine act coordinatewise. Their entries are identical within each coordinate pair, so multiplication by either vector commutes with J.

Composition of position offsets. Let $c_j = \cos(s_j\boldsymbol{\theta})$ and $t_j = \sin(s_j\boldsymbol{\theta})$ for $j \in \{1, 2\}$. After the first transformation, $k' = \bar{k} \odot c_1 + (J\bar{k}) \odot t_1$, and hence $Jk' = (J\bar{k}) \odot c_1 - \bar{k} \odot t_1$. Applying the second transformation gives

$$\begin{aligned}
\mathrm{RoPE}(k', s_2) &= k' \odot c_2 + (Jk') \odot t_2 \\
&= \bar{k} \odot c_1 \odot c_2 + (J\bar{k}) \odot t_1 \odot c_2 \\
&\quad + (J\bar{k}) \odot c_1 \odot t_2 - \bar{k} \odot t_1 \odot t_2 \\
&= \bar{k} \odot (c_1 \odot c_2 - t_1 \odot t_2) + (J\bar{k}) \odot (t_1 \odot c_2 + c_1 \odot t_2) \\
&= \bar{k} \odot \cos((s_1 + s_2)\boldsymbol{\theta}) + (J\bar{k}) \odot \sin((s_1 + s_2)\boldsymbol{\theta}) \\
&= \mathrm{RoPE}(\bar{k}, s_1 + s_2).
\end{aligned} \tag{18}$$

The penultimate equality follows from the sine and cosine addition identities. Thus $R(s_2)R(s_1) = R(s_1 + s_2)$ and $R(s)^{-1} = R(-s)$.

Correcting a cached key. Suppose $k_{p_0} = R(p_0)\bar{k}$ was cached at position p_0 and is now placed at position p. Removing the old rotation and applying the new one yields

$$\widetilde{k}_p = R(p)R(p_0)^{-1}k_{p_0} = R(p - p_0)k_{p_0} = R(p)\bar{k}. \tag{19}$$

For a query $q_u = R(u)\bar{q}$ at position u, orthogonality gives

$$q_u^\top \widetilde{k}_p = \bar{q}^\top R(u)^\top R(p)\bar{k} = \bar{q}^\top R(p - u)\bar{k}, \tag{20}$$

which restores the relative positional rotation for the new coordinates. Values and nonrotary coordinates remain unchanged.

Example. Use one-based positions and consider the assembled sequence $X[1:8] = [S; C_1; C_2]$, where the system prefix S occupies position 1, C_1 occupies positions $[2, 3, 4]$, and C_2 occupies positions $[5, 6, 7, 8]$. During independent preprocessing, each chunk follows the same prefix: $[S; C_1]$ assigns C_1 positions $[2, 3, 4]$, whereas $[S; C_2]$ assigns C_2 positions $[2, 3, 4, 5]$. When assembling their caches after a single copy of S, the positions of C_1 are unchanged. Every key of C_2 moves forward by three:

$$(p_0, p) \in \{(2, 5), (3, 6), (4, 7), (5, 8)\}, \qquad \widetilde{k}_p = R(3)k_{p_0}. \tag{21}$$

The same correction is applied at each layer and attention head, using the corresponding model's rotary parameters.

Application and scope. Target keys and selector keys use their respective models' rotations, and task queries are aligned to the same scoring coordinates as the keys they attend to. Updates use request-local buffers, preserving persistent cache entries. The constant-offset identity assumes that the old and new encodings use the same fixed frequency schedule. If the schedule changes, use $R_{\text{new}}(p)R_{\text{old}}(p_0)^{-1}$ instead. This correction changes the explicit positional rotation of a stored key; it does not reconstruct hidden-state dependencies on the new preceding context and therefore does not make independently cached states equivalent to a full prefill of the assembled prompt.

D.3 Selection Across Different Tokenizers

The selector chooses text regions; the target recomputes its own tokens covering those regions. Figure 13 illustrates this transfer. All cached and recomputed KV states belong to the target.

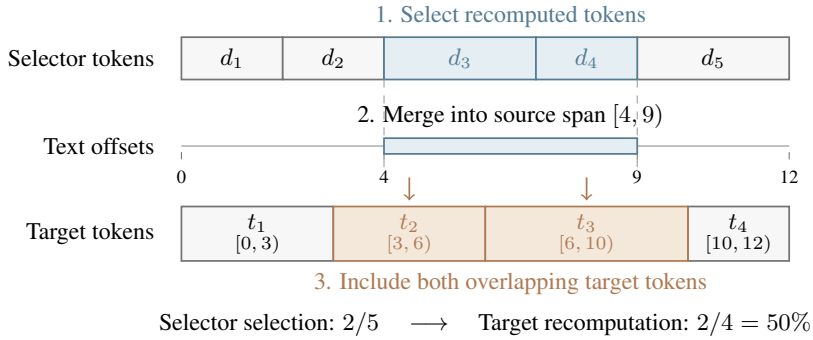

Figure 13: **Mapping selection through shared text.** Illustrative token boundaries; all four target tokens are eligible ($|\mathcal{R}| = 4$). The two selected selector tokens cover $[4, 9)$. Both overlapping target tokens are recomputed in full, including their boundary portions outside the selected span. Each target token counts once.

A common text coordinate system. Let $\mathcal{T}$ be the serialized source text of the assembled prompt X, and let L be its length in a fixed offset unit, either bytes or characters. All offsets below refer to this same text and unit, before any tokenizer-specific normalization. Let n be the number of target tokens in X, and let $\mathcal{J}_{\text{d}} \subseteq \{1, \dots, n_{\text{d}}\}$ index the assembled cached selector tokens with valid source-text offsets. For $j \in \mathcal{J}_{\text{d}}$, write $D_j = [a_j, b_j)$; for an eligible target position $i \in \mathcal{R} \subseteq \{1, \dots, n\}$, write $E_i = [c_i, d_i)$, with $0 \le a_j < b_j \le L$ and $0 \le c_i < d_i \le L$. These intervals describe text covered by tokens, not token indices or vocabulary IDs.

Target offsets are obtained for the complete prompt. Selector offsets refer to the tokens that produced the cached keys. If a cached chunk occurrence begins at offset o_g in $\mathcal{T}$ and its selector token has chunk-local interval $[\hat{a}_j, \hat{b}_j)$, its assembled interval is $D_j = [o_g + \hat{a}_j, o_g + \hat{b}_j)$. Each occurrence receives its own offset, even when identical text appears more than once. Any offsets reported against normalized text must first be mapped back to $\mathcal{T}$, including whitespace. Retokenizing the concatenated prompt with the selector or searching for decoded substrings would not, in general, recover the correspondence to the cached keys.

Overlap relation and candidate mapping. Define the many-to-many correspondence between target and selector tokens by

$$\mathsf{O}_{ij} = \mathbf{1}[\max(c_i, a_j) < \min(d_i, b_j)], \qquad i \in \mathcal{R},\ j \in \mathcal{J}_{\text{d}}. \tag{22}$$

Thus $\mathsf{O}_{ij} = 1$ exactly when E_i and D_j share a nonempty source-text interval. Intervals that only meet at an endpoint do not match. The relation permits one token to overlap several tokens from the other tokenizer; it does not require equal token boundaries or a one-to-one vocabulary mapping. The eligible selector positions are

$$B_{\mathcal{R}} = \bigcup_{i \in \mathcal{R}} E_i, \qquad \mathcal{R}_{\text{d}} = \{j \in \mathcal{J}_{\text{d}} : D_j \cap B_{\mathcal{R}} \neq \emptyset\} = \{j \in \mathcal{J}_{\text{d}} : \exists i \in \mathcal{R},\ \mathsf{O}_{ij} = 1\}. \tag{23}$$

The offset-based procedure assumes every eligible target position has a valid interval and at least one overlapping cached selector token. Positions without a reliable correspondence must be handled before selection, as described below.

Transferring the selected regions. The selector scores cached keys using Equation (13); the task text τ is tokenized separately for query prefill. For $0 < \rho < 1$, top-k selection returns $I_{\mathrm{d}} \subseteq \mathcal{R}_{\mathrm{d}}$ with $k = \lfloor \rho |\mathcal{R}_{\mathrm{d}}| \rfloor$. Transfer this selection by

$$B_{\mathrm{sel}} = \bigcup_{j \in I_{\mathrm{d}}} D_j, \qquad I = \{i \in \mathcal{R} : E_i \cap B_{\mathrm{sel}} \neq \emptyset\} = \{i \in \mathcal{R} : \exists j \in I_{\mathrm{d}},\ \mathsf{O}_{ij} = 1\}. \tag{24}$$

Overlapping or adjacent selected intervals may be merged without changing B_{sel}; gaps between them are not filled by the matching step. Each matched target token is selected once and recomputed in full, including any part outside B_{sel}. Restricting I to $\mathcal{R}$ prevents a selector token that straddles an eligibility boundary from selecting an ineligible target position. In Figure 13, $[4, 7)$ and $[7, 9)$ merge into $[4, 9)$, selecting t_2 and t_3.

Algorithm 3 specifies the two alignment steps used by Algorithm 2. Its pairwise overlap tests express the matching rule directly; the relation need not be materialized as a dense matrix. For fixed candidate sets and offsets, enlarging I_{d} can only enlarge I.

Algorithm 3 Cross-tokenizer selection through source-text overlap

Input: Eligible target positions $\mathcal{R}$ with intervals $E_i = [c_i, d_i)$; valid cached selector intervals $D_j = [a_j, b_j)$, $j \in \mathcal{J}_{\mathrm{d}}$; scores s_ϕ; requested ratio $\rho \in [0, 1]$
Output: Selected target positions $I \subseteq \mathcal{R}$
Precondition: Offsets use the same source text and unit; each $i \in \mathcal{R}$ overlaps at least one valid selector interval.

if $\mathcal{R} = \emptyset$ or $\rho = 0$ **then**
  return $\emptyset$
end if
if $\rho = 1$ **then**
  return $\mathcal{R}$
end if
$\mathcal{R}_{\mathrm{d}} \leftarrow \emptyset$ ▷ ALIGNTOSELECTOR
for each $j \in \mathcal{J}_{\mathrm{d}}$ **do**
  if $\exists i \in \mathcal{R} : \max(c_i, a_j) < \min(d_i, b_j)$ **then**
    $\mathcal{R}_{\mathrm{d}} \leftarrow \mathcal{R}_{\mathrm{d}} \cup \{j\}$
  end if
end for
$k \leftarrow \lfloor \rho |\mathcal{R}_{\mathrm{d}}| \rfloor$
$I_{\mathrm{d}} \leftarrow \textsc{TopK}(s_\phi, \mathcal{R}_{\mathrm{d}}, k)$
$B_{\mathrm{sel}} \leftarrow \textsc{MergeIntervals}(\{D_j : j \in I_{\mathrm{d}}\})$ ▷ Merge overlap or adjacency; preserve gaps
$I \leftarrow \emptyset$ ▷ ALIGNTOTARGET
for each $i \in \mathcal{R}$ **do**
  if $\exists [u, v) \in B_{\mathrm{sel}} : \max(c_i, u) < \min(d_i, v)$ **then**
    $I \leftarrow I \cup \{i\}$ ▷ Select the whole token, once
  end if
end for
return I

Control tokens and cache boundaries. Tokens with no nonempty source-text interval, including model-specific control tokens and prompt wrappers, cannot be matched by Equation (22). They require an explicit role or segment correspondence outside this interval-based procedure. A target position lacking a reliable correspondence is conservatively assigned to $\mathcal{U}$ and excluded from $\mathcal{R}$; mandatory and uncached positions are also in $\mathcal{U}$. If assembling chunks changes a target token's identity at a boundary, its old KV row is incompatible, so that position likewise enters $\mathcal{U}$ before selection. The sets $\mathcal{U}$ and $\mathcal{R}$ are disjoint. Text-span alignment transfers a selection mask only: it neither transfers selector KV states to the target nor replaces RoPE correction of compatible target keys.

Realized recomputation ratio. The requested ρ controls selector-side top-k selection. Different token boundaries and boundary coverage can change the resulting target count, so the reported ratio

for a nonempty candidate set is

$$\rho_{\mathrm{target}} = \frac{|I|}{|\mathcal{R}|}, \qquad |\mathcal{S}| = |\mathcal{U}| + |I|. \tag{25}$$

Neither $|I_{\mathrm{d}}|/|\mathcal{R}_{\mathrm{d}}|$ nor the requested ρ substitutes for this realized target-token ratio. In Figure 13, two of five selector tokens are selected (40%), but the resulting two of four eligible target tokens give $\rho_{\mathrm{target}} = 50\%$. Mandatory and uncached positions contribute to target prefill work but not to its selective-recomputation denominator. At $\rho = 0$, $I = \emptyset$; at $\rho = 1$, all eligible target positions are selected. When $\mathcal{R} = \emptyset$, selection is skipped and the ratio is not applicable.