LouisKV: Efficient KV Cache Retrieval for Long Input-Output Sequences

Anonymous authors
Paper under double-blind review

Abstract

While Key-Value (KV) cache succeeds in reducing redundant computations in auto-regressive models, it introduces significant memory overhead, limiting its practical deployment in long-sequence scenarios. Existing KV retrieval methods attempt to mitigate this by dynamically retaining only a subset of KV entries on the GPU. However, they still suffer from notable efficiency and accuracy bottlenecks due to per-token retrieval and coarse-grained page-level KV management strategy, especially in long-output reasoning scenarios. With the emergence of large reasoning models, efficiently handling such scenarios has become increasingly important. To address this issue, we present two key observations: (1) critical KVs exhibit strong temporal locality during decoding, and (2) these KVs exhibit distinct distribution patterns across the input prompt and the generated output. Building on these observations, we propose *LouisKV*, an efficient KV cache retrieval framework designed for various long-sequence scenarios. Specifically, LouisKV introduces a semantic-aware retrieval strategy that leverages temporal locality to trigger retrieval only at semantic boundaries, drastically reducing computation and data transfer overhead. LouisKV also designs a decoupled, fine-grained management scheme that tailors differentiated strategies for input and output sequences to create retrieval units that better match the model's attention patterns, thereby enabling the precise identification of critical KVs. Furthermore, to boost system efficiency, LouisKV incorporates several kernel-level optimizations, including custom Triton and CUDA kernels to accelerate the KV clustering and retrieval. Evaluation results show that LouisKV achieves up to 4.7$\times$ speedup over state-of-the-art KV retrieval methods while maintaining near-lossless accuracy across diverse long-sequence tasks, including long-input short-output, short-input long-output, and long-input long-output scenarios.

1 Introduction

Large language models (LLMs) (Achiam et al., 2023; Guo et al., 2025) have demonstrated remarkable capabilities in both long-context understanding for tasks like long document question answering (Bai et al., 2023), and in long chain-of-thought (CoT) generation for complex reasoning tasks like mathematics (Hendrycks et al., 2021). The inference process of mainstream LLMs is auto-regressive, with the KV cache (Shi et al., 2024) stored in memory to avoid recomputation. However, the memory footprint of the KV cache grows approximately linearly with the sequence length, easily exceeding GPU memory capacity.

To address this challenge, recent studies have proposed sparse attention mechanisms to reduce KV cache usage. One line of work, known as KV dropping (Xiao et al., 2023; Zhang et al., 2023; Feng et al., 2024), retains only important tokens and permanently discards those deemed less relevant. However, such methods fail to account for the dynamic nature of KV importance, that is, tokens initially considered unimportant may become critical in subsequent steps, leading to reduced inference accuracy. To address this, another line of work, KV retrieval (Chen et al., 2024a; Tang et al., 2024; Liu et al., 2024a; Chen et al., 2024b), has emerged as a promising alternative. This approach preserves the full KV cache, typically in CPU memory to avoid GPU Out-Of-Memory (OOM) errors, whereas dynamically estimating a subset of important entries and retrieving them for the current token's generation.

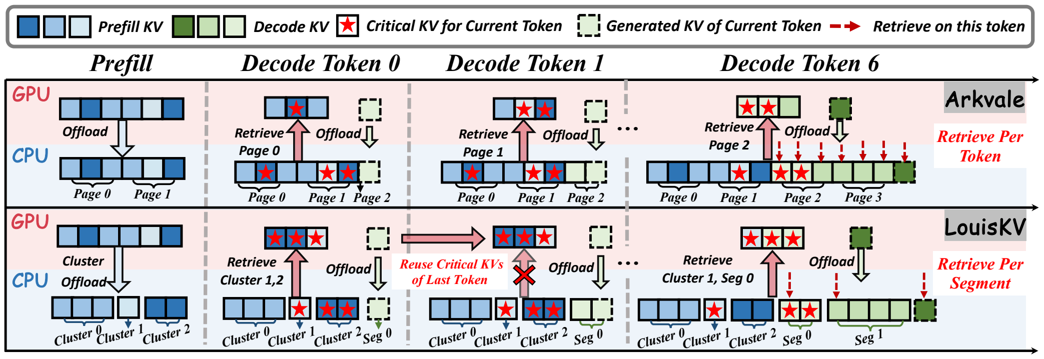


Figure 1: Comparison of Arkvale and LouisKV. Arkvale adopts page-level KV management and retrieves critical pages from the CPU for every decoding token, leading to high transfer overhead and potential accuracy degradation. In contrast, LouisKV reuses critical KVs by exploiting temporal locality to significantly reduce retrieval frequency. It also employs a decoupled KV management scheme, enabling precise retrieval to improve transfer efficiency while maintaining high accuracy.

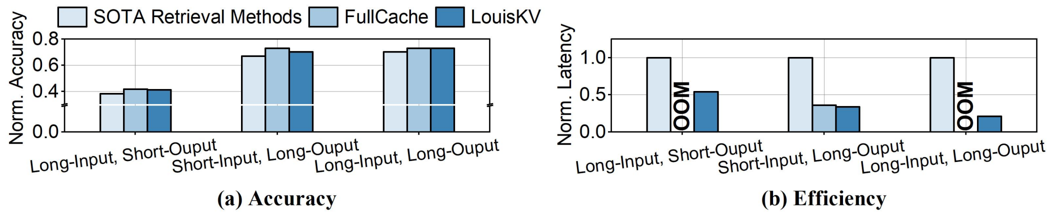


Figure 2: Accuracy and efficiency comparison on various long-sequence tasks. (a) LouisKV achieves accuracy comparable to FullCache (the lossless baseline) and superior to state-of-the-art retrieval methods. (b) LouisKV significantly reduces inference latency, substantially outperforming state-of-the-art retrieval methods, while also avoiding the Out-Of-Memory errors that FullCache faces in long-sequence and large batch size scenarios.

Although existing KV retrieval methods achieve better accuracy than KV dropping approaches by retaining complete information, these methods still face significant bottlenecks in both inference efficiency and retrieval performance. As shown in Figure 1, a primary issue is that existing methods (Zhang et al., 2025; Lee et al., 2024) trigger retrieval at every decoding step. This introduces two primary sources of overhead: the computational cost of estimating token importance and the data transfer latency of recalling KV entries from the CPU. This accumulated overhead becomes particularly prohibitive in scenarios involving long-output reasoning models (Guo et al., 2025), which have become a mainstream paradigm. Meanwhile, existing methods (Chen et al., 2024a; Xiao et al., 2024; Yuan et al., 2025) typically adopt page-level KV management strategies. However, this coarse-grained management strategy frequently transfers entire pages containing numerous non-critical KV entries, which either increase data transfer overheads or reduce inference accuracy. Therefore, enhancing inference efficiency while preserving accuracy across all long-sequence scenarios has become a key challenge.

To address this bottleneck, we conduct an empirical analysis of critical KV access patterns in long-output reasoning models (Yang et al., 2025). Our two key observations bring new insights to well address two fundamental questions: *how to trigger retrieval to reduce overhead*, and *how to manage the KV cache to improve retrieval precision*. First, we observe that *critical KVs access exhibits strong temporal locality.* Specifically, the sets of critical KV entries accessed in adjacent decoding tokens show high similarity. For instance, during mathematical reasoning, the model produces intermediate reasoning steps wherein tokens within the same step consistently attend to the same mathematical lemmas. This insight motivates triggering retrieval only at semantic boundaries. Second, we observe that *critical KVs exhibit distinct distribution patterns across input and output sequences.* Specifically, for the currently generated token, critical KVs in long input sequences are often distributed sparsely throughout the context, while those in long output sequences of reasoning models tend to concentrate locally within some reasoning steps. This inspires the adoption of different KV management strategies for the prefill and decode stages.

Building on these observations, we propose *LouisKV*, an efficient KV cache retrieval framework that integrates a semantic-aware retrieval strategy with a decoupled fine-grained management scheme, as shown in Figure 1. To exploit *temporal locality* during decoding, LouisKV groups consecutive decoding tokens into coherent segments based on query vector similarity. It triggers retrieval only at segment boundaries to drastically reduce the retrieval overhead. Based on the attention distribution, LouisKV proposes a decoupled KV management scheme tailored for distinct distribution patterns in the input and output sequence. Specifically, for the long input sequence, LouisKV employs clustering to group KV entries of semantically similar tokens into *semantic clusters*; for the long output sequence, it leverages dynamic boundaries to partition the KV cache into *temporal segments*. By creating retrieval units that better match attention distributions, LouisKV enables precise identification and transfer of critical KV entries, achieving higher accuracy and lower transfer overheads than page-level methods. Furthermore, to boost system efficiency, LouisKV incorporates several kernel-level optimizations, including custom Triton and CUDA kernels to accelerate KV clustering and retrieval. As validated by extensive experiments summarized in Figure 2, LouisKV maintains near-lossless accuracy on various long-sequence tasks while achieving up to a 4.7 $\times$ speedup in end-to-end latency compared to state-of-the-art retrieval methods.

Our contributions are as follows: (1) We conduct an empirical analysis of critical KV access patterns, revealing two key observations: strong temporal locality and distinct distribution patterns. (2) Based on these insights, we propose LouisKV, a novel KV retrieval framework that incorporates a semantic-aware retrieval strategy and a decoupled management scheme. To our knowledge, LouisKV is the first KV retrieval framework designed to comprehensively cover diverse long-sequence tasks. (3) We extensively evaluate LouisKV on various LLM benchmarks, demonstrating that it achieves a 4.7$\times$ speedup in end-to-end latency while maintaining near-lossless accuracy, as shown in Figure 2.

2 Related Work

Scaling LLMs to Long Contexts and Generations. The capability of LLMs to process long-sequence tasks is rapidly advancing. On the one hand, modern models have supported vastly longer inputs. For instance, Qwen3 (Yang et al., 2025) can handle contexts of 128K tokens, while Gemini (Team et al., 2023) models push the boundary to 2M tokens. On the other hand, a parallel trend involves generating longer outputs. Driven by test-time scaling laws, prompting models to generate more extensive, step-by-step reasoning at inference has proven to be a cost-effective method to boost model performance (Guo et al., 2025; OpenAI, 2024; Yang et al., 2025). Our work, LouisKV, targets KV cache optimization in both long-input and long-output scenarios.

KV Cache Dropping. These methods typically discard a subset of KV entries permanently based on attention scores. For instance, SnapKV (Li et al., 2024) employs a local window of recent tokens to filter the KV cache of the prompt. However, it only compresses the KV cache during prefilling and does not support long-output scenarios. In contrast, H2O (Zhang et al., 2023) retains a fixed size of KV cache during decoding by selecting tokens with the highest cumulative attention scores, while RaaS (Hu et al., 2025) evicts tokens that have not received substantial attention over a sustained period. Although these methods are computationally efficient in long-output scenarios, they often lead to significant accuracy degradation due to the irreversible loss of discarded token information.

KV Cache Retrieval. To avoid information loss inherent in KV dropping methods, KV retrieval methods preserve the complete KV cache and dynamically retrieve a critical subset for computation. However, existing approaches face several challenges. Methods like Quest (Tang et al., 2024) and SparQ (Ribar et al., 2023) encounter memory limitations when attempting to store the entire KV cache on the GPU. To overcome these GPU memory constraints, other approaches (Chen et al., 2024a; Zhang et al., 2025; Sun et al., 2024) offload the entire KV cache to CPU DRAM. However, these methods suffer from efficiency and accuracy bottlenecks. First, as illustrated in Figure 1, frequent retrieval operations for each token's generation introduce significant overhead, especially in long-output scenarios. Second, the coarse-grained management results in transferring numerous non-critical tokens, causing unnecessary data transfer overhead and even degrading model accuracy. A detailed analysis of these bottlenecks is provided in the Appendix A. In contrast, our work, LouisKV, provides a comprehensive solution that efficiently handles both long-input and long-output scenarios without compromising inference accuracy.

Semantic-Based KV Optimization. Recently, leveraging semantic information to enhance KV cache efficiency has garnered increasing attention. One line of work identifies semantic units based on natural language boundaries. For instance, SepLLM (Chen et al., 2025) compresses the information of a text segment into its corresponding separator, while SentenceKV (Zhu et al., 2025) treats full sentences as fundamental retrieval units, managing the KV cache at a coarse-grained sentence level. Another line of methods employs clustering to organize the KV cache. For example, Squeezed Attention (Hooper et al., 2025) and ClusterKV (Liu et al., 2024b) apply K-means clustering to group semantically similar key vectors from the input sequence. While these approaches demonstrate the potential of semantic-aware KV management, they typically rely on static linguistic rules or apply a uniform strategy across the entire sequence. In contrast, LouisKV distinguishes itself through two key innovations: (1) An adaptive retrieval strategy that exploits temporal locality to dynamically identify semantic boundaries during decoding; and (2) A decoupled management scheme that uniquely tailors clustering for the sparse patterns in input sequences and temporal segmentation for the dense patterns in output sequences.

3 MOTIVATION

3.1 PRELIMINARIES

The auto-regressive inference of LLMs involves two phases: prefill and decode. During prefilling, the model processes the entire input prompt to generate initial KV cache for each layer, denoted $K_{inp} \in \mathbb{R}^{P\times d}$ and $V_{inp} \in \mathbb{R}^{P\times d}$, where P is the prompt length and d is the hidden dimension. During decoding, the model generates tokens auto-regressively until encountering an end-of-sequence token. At decoding step t, the model generates a query vector $q_t \in \mathbb{R}^{1\times d}$ for each layer. Simultaneously, the corresponding key k_t and value $v_t \in \mathbb{R}^{1\times d}$ are computed and appended to the historical KV cache, forming updated K_t and V_t. The output o_t is then computed via the attention mechanism:

$$o_t = A_t V_t = \text{Softmax}\left(\frac{q_t K_t^T}{\sqrt{d}}\right) V_t$$

Here, A_t represents the attention weights indicating the importance of each historical KV entry to the current query q_t.

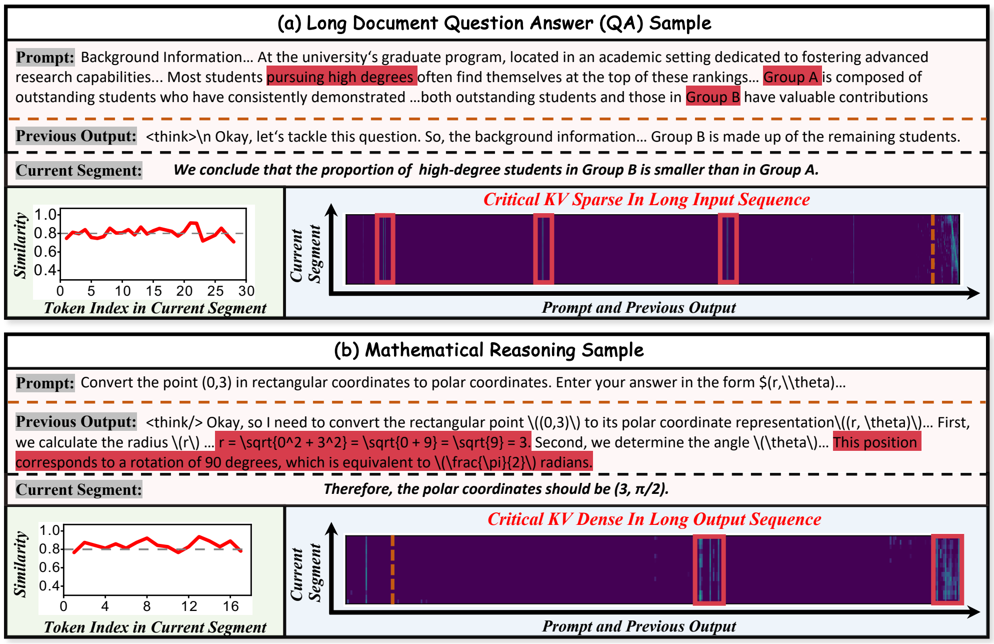
KV retrieval methods usually select a subset of KV entries under a fixed budget B. Quest (Tang et al., 2024) and Arkvale (Chen et al., 2024a) partition the entire key cache $(k_0, k_1, \ldots, k_{P+t})$ into m fixed-size pages and construct index C_i for each page i. At every decoding step t, they select the most critical pages by computing the similarity between the current query q_t and all indices $(C_0 ... C_{m-1})$. All KV entries within a selected page are retrieved for the attention computation. The selection process can be formalized as $\mathcal{I}_t = \text{Sel}(q_t, K_t)$, where $\mathcal{I}_t$ is the set of KV indices selected in step t, with the constraint $|\mathcal{I}_t| \leq B$. Ultimately, only the selected KV subset $(K_{\mathcal{I}_t}, V_{\mathcal{I}_t})$ participates in the attention computation, significantly reducing the inference latency and memory footprint.

3.2 OBSERVATIONS

We conducted an in-depth analysis of critical KV access patterns on long-sequence tasks (Jia, 2025; Ling et al., 2025), leading to two key observations that directly motivate our design. These observations provide insights into two fundamental questions: 1) When should retrieval be triggered, and 2) How should the KV cache be managed to enable precise and efficient retrieval?

Observation 1: Critical KVs access exhibits strong temporal locality during decoding. The model tends to produce its output as a sequence of semantically coherent *segments*. For instance, one reasoning step in a mathematical problem forms one segment, as illustrated in Figure 3(b). We hypothesize that within each segment, the model continuously attends to a relatively similar subset of tokens to maintain semantic coherence.

To validate this hypothesis, we calculated the Jaccard similarity between the critical KV index sets, $\mathcal{I}_t$ and $\mathcal{I}_{t+1}$, of adjacent decoding tokens within the same segment. The results provide clear evidence for our hypothesis. As shown in Figure 3(a) and (b), the similarity curves (bottom left) indicate that the Jaccard similarity remains consistently above 0.8 within *Current Segment* for both document QA and mathematical reasoning examples. This observation is also visually confirmed by the attention heatmaps (bottom right of Figure 3(a) and (b)). The bright, vertical bands in the



(a) Long Document Question Answer (QA) Sample

Prompt: Background Information... At the university's graduate program, located in an academic setting dedicated to fostering advanced research capabilities... Most students pursuing high degrees often find themselves at the top of these rankings... Group A is composed of outstanding students who have consistently demonstrated ...both outstanding students and those in Group B have valuable contributions

Previous Output: <think>\n Okay, let's tackle this question. So, the background information... Group B is made up of the remaining students.

Current Segment: *We conclude that the proportion of high-degree students in Group B is smaller than in Group A.*

(b) Mathematical Reasoning Sample

Prompt: Convert the point (0,3) in rectangular coordinates to polar coordinates. Enter your answer in the form $(r,\\theta)...

Previous Output: <think/> Okay, so I need to convert the rectangular point \((0,3)\) to its polar coordinate representation\((r, \theta)\)... First, we calculate the radius \(r\) ... r = \sqrt{0^2 + 3^2} = \sqrt{0 + 9} = \sqrt{9} = 3. Second, we determine the angle \(\theta\)... This position corresponds to a rotation of 90 degrees, which is equivalent to \(\frac{\pi}{2}\) radians.

Current Segment: *Therefore, the polar coordinates should be (3, π/2).*

Similarity
1.0
0.8
0.6
0.4
0 5 10 15 20 25 30
Token Index in Current Segment
Critical KV Sparse In Long Input Sequence
Current Segment
Prompt and Previous Output
Similarity
1.0
0.8
0.6
0.4
0 4 8 12 16
Token Index in Current Segment
Critical KV Dense In Long Output Sequence
Current Segment
Prompt and Previous Output

Figure 3: Access patterns of critical KVs in different long-sequence tasks. First, both tasks demonstrate temporal locality: during the generation of a coherent segment (Current Segment), the similarity of critical KV sets maintains high values. Second, they reveal a distinct spatial distribution: (a) in a long-document task, attention is sparsely distributed in the prompt, whereas (b) in a mathematical reasoning task, attention is densely focused on some intermediate steps in the previous output.

heatmaps indicate that attention is persistently focused on the same regions of the prompt and previous output across the iterations of the current segment's generation. The observed temporal locality renders per-token KV retrieval unnecessary, motivating a semantic-aware strategy that amortizes the retrieval overhead across multiple tokens.

Observation 2: Critical KVs exhibit distinct distribution patterns across the input and output sequences. While temporal locality informs when to retrieve, the spatial distribution of critical KVs informs how to retrieve critical KV entries both efficiently and precisely.

As exemplified by the long-document QA task in Figure 3(a), we observe that critical KVs in the long input sequence are distributed sparsely. Specifically, the attention heatmap reveals that high-attention regions are sparsely dispersed across the long input sequence. This shows that the model must extract scattered information from distant parts of the context to generate the answer. In contrast, critical KVs in a long output sequence are often concentrated densely. As shown in Figure 3(b), for tasks like mathematical reasoning, the model's attention focuses densely on the previously generated segments, such as intermediate lemmas. As discussed in Section 3.1, existing KV retrieval methods typically employ fixed-size pages as the fundamental unit for management and selection. However, this approach ignores the distinct spatial distribution of critical KVs, leading to the retrieved pages containing numerous non-critical KV entries. This distinct distribution pattern poses a fundamental challenge for conventional page-level management mechanisms. This strongly motivates the design of a finer-grained KV management mechanism capable of adapting to this distinct distribution.

4 LOUISKV SYSTEM

4.1 SEMANTIC-AWARE ADAPTIVE RETRIEVAL

As established in Observation 1, the model tends to focus on a highly similar KV subset across multiple consecutive tokens during decoding, rendering per-token retrieval redundant and inefficient. To leverage this temporal locality, LouisKV introduces a semantic-aware adaptive retrieval strategy

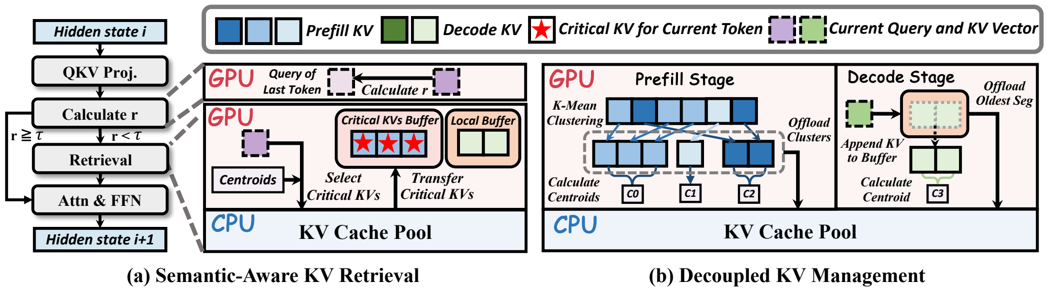


Figure 4: The design of LouisKV. (a) Retrieval is triggered when the query similarity r drops below a threshold τ, loading critical KV entries from the CPU cache pool. (b) During prefilling, K-means clustering is employed to group semantically similar KVs into clusters. During decoding, consecutively generated KVs are partitioned into temporal segments. These clusters and segments are then offloaded to a unified cache pool on the CPU. The detailed algorithm is provided in Appendix C.

that operates at the granularity of *temporal segments*, thereby amortizing the retrieval cost over multiple decoding tokens.

The central challenge is to identify the boundaries of these segments accurately with low overhead. A naive approach is to compare the critical KV index sets of adjacent tokens. However, this approach incurs substantial computational costs, as the Sel function itself involves dense matrix computations. LouisKV therefore employs a lightweight alternative (Xu et al., 2024; Liu et al., 2025). Specifically, for each transformer layer at each decoding step t, LouisKV computes the cosine similarity between the current query vector q_t and the previous one q_{t-1}, averaged across all attention heads:

$$r_t = \frac{1}{H}\sum_{h=1}^{H} \text{cosine}(q_{t-1}^h, q_t^h)$$

where H is the total number of attention heads and q_t^h is the query vector for head h at step t. As shown in Figure 4(a), when this similarity score r drops below a predefined threshold τ, LouisKV identifies a semantic boundary and triggers a KV retrieval operation. This operation selects the critical units that scored the highest by computing the similarity between the current query q_t and the centroids C of all units, and subsequently loads the corresponding KV cache from CPU to GPU.

4.2 Decoupled Fine-grained Management

After determining when to trigger the retrieval, the subsequent challenge is how to manage the KV cache to improve retrieval precision. According to Observation 2, traditional coarse-grained page-level management failed to efficiently adapt to the distinct distribution patterns across the input and output sequences. LouisKV therefore designs a decoupled fine-grained management scheme (Figure 4(b)) that tailors retrieval units separately for each phase.

Prefill Stage. To manage the sparse critical KVs in the input sequence, LouisKV introduces a cluster-granularity strategy. Since the attention score for a query vector q_t is primarily determined by its dot product with each key vector, key vectors that are similar in semantic space tend to exhibit similar attention scores (Liu et al., 2024b). Accordingly, LouisKV employs the k-means clustering algorithm to group similar key vectors into the same cluster. Subsequently, LouisKV calculates the centroid C_i for each cluster by averaging all its key vectors. Unlike traditional page-level strategies, the key vectors within the same cluster are semantically similar, allowing this centroid to serve as a more precise index for subsequent retrieval. This KV clustering and offloading process occurs asynchronously with the prefill forward pass, avoiding blocking the forward computation.

Decode Stage. To handle the highly dense attention pattern in the output sequence, LouisKV leverages the semantic boundaries identified in Section 4.1 to partition the generated KV cache into multiple temporal segments. Similar to the semantic clusters from the prefill stage, each temporal segment is represented by an index vector derived from averaging its key vectors. To prevent excessive GPU memory consumption during decoding, LouisKV maintains a fixed-size local buffer on the GPU to cache the most recently generated KV segments. When this buffer becomes full, the oldest segment will be offloaded to CPU memory. These offloaded temporal segments from the

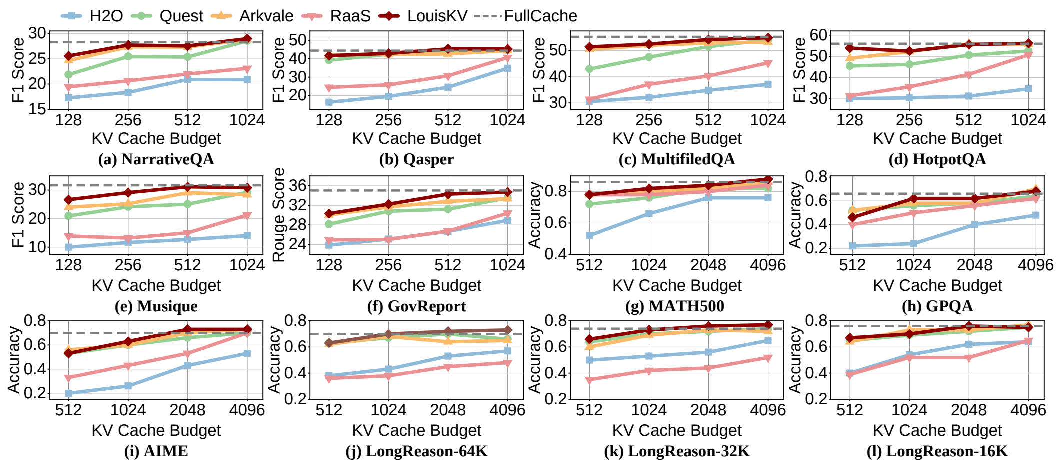


Figure 5: Performance comparison of LouisKV against four baseline methods across various cache budgets. Subplots (a-f) present the results on six long-input understanding tasks using Llama-3.1-8B-Instruct. Subplots (g-l) display the results on six long-output reasoning tasks using Qwen3-8B. For detailed experimental results on more models, please refer to Appendix D.1.

decode stage, together with the semantic clusters from the prefill stage, form a unified KV cache pool in CPU memory.

4.3 System Optimizations

To enhance overall system efficiency, LouisKV incorporates several system-level and kernel-level optimizations. First, we implement a custom Triton (Tillet et al., 2019) kernel to efficiently execute the clustering operation during prefilling. Second, we introduce a group-consistent selection strategy that selects a unified set of KV clusters and segments for an entire query group, which efficiently adapts to the Grouped-Query Attention (Ainslie et al., 2023) by avoiding redundant data transfers (see Appendix B for details). Finally, since these clusters or segments contain a variable number of KV entries, selecting critical items under a fixed budget in standard frameworks like PyTorch is complex and inefficient. To address this, we design a highly optimized CUDA kernel that supports rapid, batched selection of critical KVs under a given budget. Furthermore, to facilitate efficient transfer of selected KV entries between CPU and GPU, we use the DGL library (Wang, 2019) to directly transfer specific rows from CPU tensors to the GPU device, rather than first gathering the rows on the CPU and then transferring them.

5 Evaluation

5.1 Experimental setup

Datasets and Models. To comprehensively evaluate LouisKV, we select the following datasets: (1) *Long-Input, Short-Output.* This type of task primarily involves document understanding. Thus, we select six representative and popular datasets from the LongBench (Bai et al., 2023) benchmark: NarrativeQA, Qasper, MultiFieldQA, HotpotQA, Musique, and GovReport. (2) *Short-Input, Long-Output.* We chose three challenging mathematical and scientific reasoning datasets, sampling 50 problems from each: MATH500 (Hendrycks et al., 2021) and AIME (Jia, 2025), which require multi-step mathematical problem-solving, and GPQA (Rein et al., 2024), which comprises graduate-level questions demanding complex reasoning. (3) *Long-Input, Long-Output.* We choose the LongReason (Ling et al., 2025) dataset. This dataset synthetically extends short-context reasoning problems into complex versions requiring long-range dependency, and we sample 100 problems from each of its 16K, 32K, and 64K input-length subsets, denoted as LongReason-16K, LongReason-32K, and LongReason-64K, respectively. Our evaluation is conducted on two models: Qwen3-8B (Yang et al., 2025) and Llama-3.1-8B-Instruct (Grattafiori et al., 2024). We leverage the thinking mode of Qwen3-8B to handle long-output tasks requiring complex reasoning, while Llama-3.1-8B-Instruct addresses traditional long-input, short-output tasks.

Baselines. We compare LouisKV against state-of-the-art KV cache optimization methods, including KV dropping methods such as H2O (Zhang et al., 2023) and RaaS (Hu et al., 2025), KV retrieval methods such as Quest (Tang et al., 2024) and Arkvale (Chen et al., 2024a). We set a uniform KV cache budget B for all methods. Following the setting of Quest, we preserve the complete KV cache for the first two layers and retain the sink tokens S and a local buffer W on the GPU across all methods, where the values of S and W are adjusted according to the specific task. Furthermore, to maintain a fair comparison, we adapt the implementations of RaaS, Quest, and Arkvale to be group-consistent. For method-specific parameters, we set the page size to 16 for Quest, Arkvale, and RaaS. For our proposed LouisKV, we set the average cluster size to 16 for the input sequence and use varying similarity thresholds τ for the output sequence, depending on the model and task.

Environment. Our server is equipped with a single NVIDIA A6000 GPU (48 GB) and an AMD Ryzen 9 5950X 16-Core Processor. The CPU and GPU are interconnected via PCIe 4.0 ×16 (32GB/s). All experiments run on Ubuntu 20.04 with Linux kernel 5.4.0 and CUDA 12.4.

5.2 Accuracy evaluation

Long-Input Understanding Tasks. We follow the methodology of Quest by simulating the decoding process, feeding the question part to the model token-by-token. We configure $S = 32$, $W = 512$, and $\tau = 0.85$, where the window size W is set to be large enough to fully retain the KV cache of all generated tokens in these short-output tasks. Figure 5(a-f) illustrates the superiority of LouisKV on long-input tasks. The results show that KV retrieval methods, including Quest and Arkvale, generally outperform KV dropping methods like H2O and RaaS, which confirms that permanently discarding the KV cache leads to accuracy degradation. More importantly, LouisKV surpasses other page-level management methods on nearly all long-input tasks under the same KV cache budget. Under a budget of 512, LouisKV achieves an average accuracy improvement of 1.1% over Arkvale and 3.3% over Quest. The superior performance of LouisKV further validates the effectiveness of our clustering management designed for the input sequence. Unlike coarse-grained page-level management, our approach more precisely identifies and retains the most critical KVs, thereby achieving higher accuracy under limited budgets.

Long-Output Reasoning Tasks. For long-output tasks that demand complex reasoning, we apply different parameter configurations. In short-input, long-output tasks, we set $S = 500$, $W = 128$, and $\tau = 0.7$, where S is configured to ensure the short prompt's KV cache is fully retained for inference, without exceeding the actual input length. In long-input, long-output tasks, we set $S = 64$, $W = 256$, and $\tau = 0.7$. The maximum generation length for all long-output tasks is set to 16K tokens, except for AIME, which is set to 32K. As depicted in Figure 5(g-l), LouisKV achieves the best or second-best accuracy in all long-output reasoning tasks under various budgets. The accuracy degradation of KV dropping methods is particularly pronounced on highly challenging tasks like AIME and long-input tasks like LongReason. Specifically, under a budget of 1024, the average accuracy of LouisKV is 2.0% higher than Arkvale and 4.8% higher than Quest across these tasks. These results ultimately validate the effectiveness of our decoupled KV management scheme for input and output sequences.

5.3 Efficiency evaluation

We evaluate the inference efficiency of LouisKV in comparison with FullCache and Arkvale. Specifically, the FullCache baseline, which retains the entire KV cache on GPU, is implemented using FlashAttention-2 (Dao, 2023). The evaluation covers three long-sequence patterns: long-input, short-output (32K input, 512 output); short-input, long-output (500 input, 16K output); and long-input, long-output (32K input, 16K output). Following the experimental setup from the Section 5.2, we use Llama-3.1-8B-Instruct for long-input scenarios and Qwen3-8B for long-output scenarios.

End-to-End Latency. As illustrated in Figure 6, compared to the state-of-the-art KV retrieval method Arkvale, LouisKV achieves up to 1.9×, 2.9× and 4.7× speedups across the three patterns, respectively. This efficiency gain stems from our semantic-aware retrieval strategy, which significantly reduces computation and data transfer overhead, and optimized kernels, which enhance the efficiency of importance estimation and data transfer. In particular, this advantage becomes more pronounced as the batch size increases, because larger batches exacerbate the data transfer bottleneck. LouisKV exhibits comparable latency compared with FullCache. When processing long sequences with larger batches, FullCache encounters OOM errors due to its significant memory

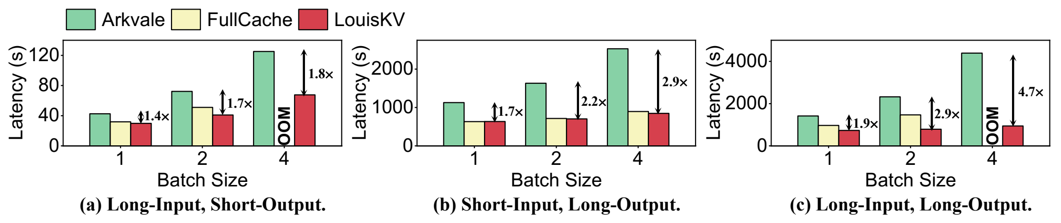


Figure 6: End-to-end latency comparison under different batch sizes and long-sequence scenarios.

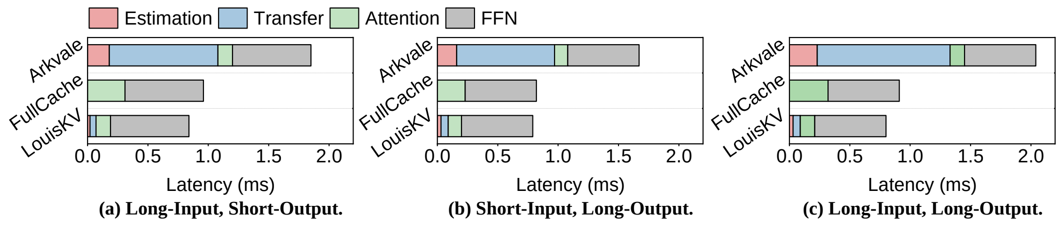


Figure 7: Per-layer latency breakdown during the decoding phase across the three scenarios.

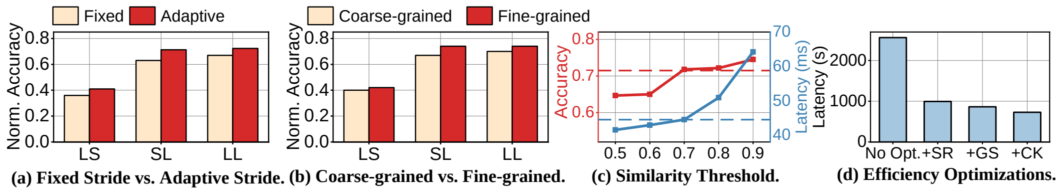


Figure 8: Ablation studies. (a) Accuracy comparison of our adaptive stride retrieval against a fixed stride baseline. (b) Accuracy comparison of our decoupled, fine-grained management against a coarse-grained baseline. (c) Impact of the similarity threshold τ on accuracy and latency. (d) Efficiency contribution of individual system optimizations.

footprint. In contrast, LouisKV offloads the entire KV cache to CPU memory, enabling support for larger batch sizes, thereby achieving higher system throughput.

Latency Breakdown. We evaluate the latency breakdown for a single transformer layer during decoding. As shown in Figure 7, although Arkvale retrieves only a subset of the KV cache, the overhead associated with retrieval, including both estimation and transfer, still accounts for 65% of the total latency, highlighting the necessity to reduce this overhead. LouisKV drastically reduces this overhead, lowering its contribution to just 11% of the total latency. We also provide the analyses of memory efficiency in Appendix D.2.

5.4 Ablation Study

Effectiveness of Core Strategies. We independently validate the effectiveness of the two core strategies in LouisKV. First, we evaluate the adaptive retrieval against a baseline that employs a fixed-stride retrieval. This result in Figure 8(a) shows that our strategy consistently achieves higher accuracy in all tasks. Furthermore, we assess the effectiveness of our fine-grained management by comparing it to a coarse-grained baseline with page-level management. The results in Figure 8(b) show that our management scheme consistently outperforms the baseline. A detailed analysis of this ablation study is provided in Appendix D.3.

Impact of Similarity Threshold τ. The threshold τ is a key hyperparameter that governs the trade-off between accuracy and efficiency. We explore the impact of τ on the long-output reasoning tasks. As shown in Figure 8(c), a higher τ leads to more frequent retrievals, thus improving inference accuracy. However, this comes at the cost of higher latency due to the overhead of retrieval. The results show that for Qwen3-8B, $\tau = 0.7$ represents an ideal balance, achieving significant efficiency gains while maximally preserving inference accuracy. Further details are in Appendix D.4.

Impact of System Optimizations. Finally, we quantify the efficiency contribution of the system optimizations in LouisKV, testing on the Qwen3-8B model in long-input, long-output scenarios.

As shown in Figure 8(d), we incrementally apply each optimization on top of a baseline system. Semantic-Aware Retrieval (SR) contributes the largest performance improvement, achieving a nearly 2.6× speedup by drastically reducing the overhead from redundant retrieval and data loading operations. Building on this, Group-Consistent Selection (GS) and our Custom Retrieval Kernel (CK) provide additional performance gains of 13.1% and 15.7%, respectively.

6 Conclusion

In this paper, we propose LouisKV, an effective KV retrieval framework designed for various long-sequence scenarios. Our work is motivated by two key observations: the strong temporal locality of critical KVs, and their distinct distribution patterns across input and output sequences. Building on these insights, LouisKV introduces a semantic-aware adaptive retrieval strategy and a decoupled management scheme. Evaluation results show that LouisKV achieves up to a 4.7 $\times$ speedup compared to state-of-the-art KV retrieval methods while maintaining near-lossless inference accuracy.

7 Ethics Statement

Our research is conducted in full compliance with the ICLR Code of Ethics. We are committed to upholding the highest standards of academic integrity and have addressed key ethical considerations as follows:

- **Research Integrity:** We have upheld research integrity by ensuring all data, methods, and results are presented truthfully and without misrepresentation. Our methodology is described with sufficient detail to support independent verification and reproducibility.
- **Data and Privacy:** The datasets utilized in our experiments are publicly available and were handled in strict accordance with their respective licenses and usage terms. Our study did not involve human subjects or personally identifiable information.
- **Attribution and Contribution:** Proper attribution has been given to all prior work through comprehensive citations. All authors listed have made significant intellectual contributions to this research and have collectively approved this submission.
- **Societal Impact:** We have considered the potential societal impacts of our work. While the intended applications are beneficial, we acknowledge the possibility of misuse and advocate for the responsible development and deployment of our findings.

8 Reproducibility statement

To ensure the reproducibility of our results, we have provided detailed descriptions of our experimental setup, hyperparameters, and implementation details. We plan to make the source code publicly available upon publication to facilitate verification and future research.

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.

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Guoxuan Chen, Han Shi, Jiawei Li, Yihang Gao, Xiaozhe Ren, Yimeng Chen, Xin Jiang, Zhenguo Li, Weiyang Liu, and Chao Huang. Sepllm: Accelerate large language models by compressing one segment into one separator, 2025. URL `https://arxiv.org/abs/2412.12094`.

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. Arkvale: Efficient generative llm inference with recallable key-value eviction. *Advances in Neural Information Processing Systems*, 37:113134–113155, 2024a.

Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. Magicpig: Lsh sampling for efficient llm generation. *arXiv preprint arXiv:2410.16179*, 2024b.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *arXiv preprint arXiv:2407.11550*, 2024.

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.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Monishwaran Maheswaran, Sebastian Zhao, June Paik, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. Squeezed attention: Accelerating long context length llm inference, 2025. URL `https://arxiv.org/abs/2411.09688`.

Junhao Hu, Wenrui Huang, Weidong Wang, Zhenwen Li, Tiancheng Hu, Zhixia Liu, Xusheng Chen, Tao Xie, and Yizhou Shan. Raas: Reasoning-aware attention sparsity for efficient llm reasoning. *arXiv preprint arXiv:2502.11147*, 2025.

Maxwell Jia. `https://huggingface.co/Maxwell-jia/aime_2024`, February 2025.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, 2024.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Zhan Ling, Kang Liu, Kai Yan, Yifan Yang, Weijian Lin, Ting-Han Fan, Lingfeng Shen, Zhengyin Du, and Jiecao Chen. Longreason: A synthetic long-context reasoning benchmark via context expansion. *arXiv preprint arXiv:2501.15089*, 2025.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *arXiv preprint arXiv:2409.10516*, 2024a.

Guangda Liu, Chengwei Li, Jieru Zhao, Chenqi Zhang, and Minyi Guo. Clusterkv: Manipulating llm kv cache in semantic space for recallable compression. *arXiv preprint arXiv:2412.03213*, 2024b.

Guangda Liu, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Danning Ke, Minyi Guo, and Jieru Zhao. Freekv: Boosting kv cache retrieval for efficient llm inference. *arXiv preprint arXiv:2505.13109*, 2025.

OpenAI. o1 system card, 2024.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. Sparq attention: Bandwidth-efficient llm inference. *arXiv preprint arXiv:2312.04985*, 2023.

Luohe Shi, Hongyi Zhang, Yao Yao, Zuchao Li, and Hai Zhao. Keep the cost down: A review on methods to optimize llm's kv-cache consumption. *arXiv preprint arXiv:2407.18003*, 2024.

Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. Shadowkv: Kv cache in shadows for high-throughput long-context llm inference. *arXiv preprint arXiv:2410.21465*, 2024.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. *Advances in Neural Information Processing Systems*, 37:119638–119661, 2024.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2023.

Fangyuan Xu, Tanya Goyal, and Eunsol Choi. Refreshkv: Updating small kv cache during long-form generation. *arXiv preprint arXiv:2411.05787*, 2024.

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.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. Pqcache: Product quantization-based kvcache for long context llm inference. *Proceedings of the ACM on Management of Data*, 3(3):1–30, 2025.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

Yuxuan Zhu, Ali Falahati, David H. Yang, and Mohammad Mohammadi Amiri. Sentencekv: Efficient llm inference via sentence-level semantic kv caching, 2025. URL `https://arxiv.org/abs/2504.00970`.

A Detailed Bottleneck Analysis of KV Retrieval Methods

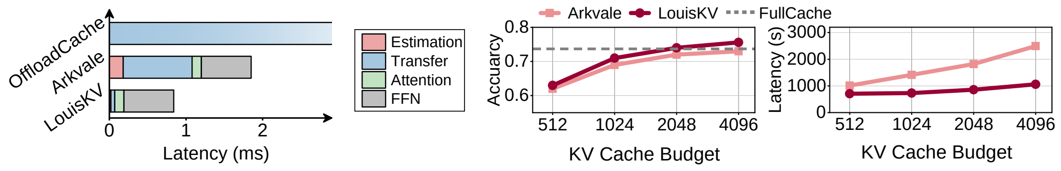


Figure 9: Bottleneck analysis of KV retrieval methods. Left: Per-layer latency breakdown comparing LouisKV with other methods. OffloadCache serves as a baseline that preloads the full KV cache of the next layer from CPU during inference, while Arkvale is a representative state-of-the-art retrieval method. Right: Inference accuracy (middle) and end-to-end latency (right) of LouisKV and Arkvale in long-output scenarios under different KV cache budgets.

While existing KV retrieval methods avoid GPU Out-Of-Memory errors by offloading the entire KV cache to CPU memory, they still face two fundamental bottlenecks regarding inference efficiency and retrieval accuracy.

First, the per-token retrieval strategy incurs prohibitive cumulative overhead. As illustrated in Figure 9 (left), although retrieval methods like Arkvale attempt to reduce KV cache transfer costs by loading less data, they introduce an additional computational overhead to estimate the importance of each page. Meanwhile, due to the limited CPU-GPU bandwidth, the data transfer overhead continues to be a primary performance bottleneck. While this overhead might seem minor for a single token generation, it accumulates linearly as each decoding token in a long-output sequence requires a new retrieval operation. This cumulative effect becomes substantial in long-output scenarios, leading to a dramatic increase in total inference latency, as shown in Figure 9 (right) for Arkvale.

Second, the coarse-grained page-level management scheme leads to suboptimal retrieval performance, creating a dilemma between efficiency and accuracy. These methods partition the sequence into fixed-size pages and use the page as the minimum unit for retrieval, which introduces a difficult trade-off. On the one hand, it results in a significant accuracy drop under tight KV cache budgets, as shown in Figure 9 (middle). On the other hand, allocating a larger budget to prevent this accuracy loss results in significantly higher end-to-end latency, as depicted in Figure 9 (right). In contrast, by retrieving the critical KVs precisely, LouisKV maintains high inference accuracy even under a small budget while simultaneously achieving high efficiency.

B Details of System Optimization

Group-consistent Selection. Existing retrieval methods are primarily designed for Multi-Head Attention architectures, where the number of query heads h_q equals the number of KV heads h_{kv}. However, modern models often adopt Grouped-Query Attention, in which a group of $g = h_q/h_{kv}$ query heads share the same KV head. To avoid excessive transfer overhead caused by retrieving distinct KV heads for each query head, we propose a group-consistent retrieval strategy. This approach selects a common set of KV indices for all queries within a group, guided by a group-aggregated score A_t^i, computed by averaging the softmax-normalized attention scores from each query:

$$A_t^i = \frac{1}{g}\sum_{j=1}^{g} \text{softmax}\left(\frac{q_t^{i*g+j}(C^i)^T}{\sqrt{d}}\right)$$

where q_t^{i*g+j} is the query of the j-th head within group i at step t, and C^i is the matrix containing the centroid vectors of all clusters and segments belonging to the i-th KV head, the one shared by the i-th query group. Based on this aggregated score, LouisKV ranks all KV clusters and segments and selects the top-scoring units. This strategy ensures that all requests from the group are unified into a single, coalesced data transfer, perfectly aligning with the computational pattern of the GQA architecture and fundamentally eliminating the transfer overhead. Notably, for conventional MHA architectures where the group size $g = 1$, our group-aware strategy naturally simplifies to a standard per-head retrieval mechanism, thus providing a unified approach for different attention structures.

C Algorithm Overview

Algorithm 1 The LouisKV Algorithm

```
1: **Input:** Model $M$, Input Sequence $X_{inp}$, KV Cache Budget $B$, Similarity Threshold $r$, A KV
   Manager $kvm$
2: **Output:** Generated Token Sequence $O$

3: function KVM.STORE_CACHE($K, V$, stage)
4:     if stage == 'prefill' then
5:         $K_{clusters} \leftarrow$ KMeans($K$)
6:         for each cluster $c$ in $K_{clusters}$ do
7:             $centroid \leftarrow$ ComputeCentroid($c$);
8:             Store $centroid$ on GPU;
9:         end for
10:        Offload $(K, V)$ to CPU memory pool asynchronously
11:    else if stage == 'decode' then
12:        $KV_{local} \leftarrow [KV_{local} : (k_t, v_t)]$
13:        if $KV_{local}$ is full then
14:            $K_{seg}, V_{seg} \leftarrow$ the oldest KV segment in $KV_{local}$
15:            $centroid \leftarrow$ ComputeCentroid($K_{seg}$);
16:            Store $centroid$ on GPU
17:            Offload $(K_{seg}, V_{seg})$ to CPU memory pool asynchronously.
18:        end if
19:    end if
20: end function

21: function KVM.RETRIEVE($q_t, B$)
22:     $Scores \leftarrow$ Compute similarity scores between query $q_t$ and all centroids
23:     $TopIndices \leftarrow$ Select the set of top-scoring clusters and segments based on $Scores$ and budget $B$
24:     $KV_{critical} \leftarrow$ Load the corresponding KVs of $TopIndices$ from CPU memory pool
25:     return $KV_{critical}$
26: end function

27: procedure LOUISKV($M, X_{inp}, B, r, kvm$)

28:     // Phase 1: Prefill Stage
29:     $K_{inp}, V_{inp} \leftarrow$ Process $X_{inp}$ with model $M$ to get initial KV cache
30:     $kvm$.store_cache($K_{inp}, V_{inp}$, 'prefill')

31:     $O \leftarrow \emptyset$; $q_{prev} \leftarrow$ None;
32:     // Phase 2: Decode Stage
33:     for $t = 1, 2, \ldots, T$ do
34:         $q_t, k_t, v_t \leftarrow$ Generate query and KV cache from the previous token's state
35:         is_new_segment $\leftarrow (t == 1) \vee$ (CosineSimilarity$(q_t, q_{prev}) < r$); $q_{prev} \leftarrow q_t$
36:         if is_new_segment == True then
37:             $KV_{critical} \leftarrow kvm$.retrieve( $q_t$, $B$ )
38:             $kvm$.store_cache($k_t, v_t$, 'decode'); $KV_{attn} \leftarrow [KV_{critical} : KV_{local}]$
39:         else if is_new_segment == False then
40:             $kvm$.store_cache($k_t, v_t$, 'decode'); $KV_{attn} \leftarrow [KV_{critical} : KV_{local}]$
41:         end if
42:         $o_t \leftarrow$ Model $M$ Generate token using $q_t$ and $KV_{attn}$; $O$.append($o_t$);
43:     end for
44:     return $O$
45: end procedure
```

D Detailed Experimental Results and Analysis

D.1 Accuracy Evaluation

Methods	Long-Input, Short-Output Tasks					
	NarrativeQA	Qasper	MultifieldQA	HotpotQA	Musique	GovReport
Llama3.1-8B-Instruct	28.26	44.38	55.27	55.97	31.68	35.02
H2O	20.90	24.50	34.82	31.29	12.70	26.69
RaaS	22.01	30.72	40.31	41.54	15.02	26.74
Quest	25.35	43.65	51.49	51.63	25.09	32.20
Arkvale	27.32	42.59	52.88	**55.92**	29.07	32.79
LouisKV	**27.53**	**45.31**	**54.22**	55.64	**31.20**	**33.30**
Qwen2.5-7B-Instruct	20.1	39.29	46.66	55.84	25.88	31.4
H2O	7.36	20.04	31.07	30.75	15.29	22.48
RaaS	10.7	25.21	35.90	39.43	15.83	26.2
Quest	15.41	34.56	44.75	45.58	22.51	29.21
Arkvale	17.64	**38.78**	44.40	**53.16**	25.81	**30.6**
LouisKV	**19.39**	37.67	**46.63**	52.14	**28.51**	30.57

Table 1: Accuracy comparison on Long-Input Understanding tasks with a KV cache budget of 512 tokens across all methods. The best results for each model family (excluding the baseline) are in **bold**.

Methods	Short-Input, Long-Output Tasks			Long-Input, Long-Output Tasks		
	MATH500	GPQA	AIME	LongReason-64K	LongReason-32K	LongReason-16K
Qwen3-8B	0.86	0.66	0.70	0.70	0.74	0.76
H2O	0.66	0.24	0.26	0.43	0.53	0.54
RaaS	0.78	0.50	0.43	0.38	0.42	0.52
Quest	0.76	0.56	0.60	0.67	0.70	0.73
Arkvale	0.82	0.60	0.63	0.68	0.68	**0.75**
LouisKV	**0.86**	**0.62**	**0.66**	**0.70**	**0.73**	0.74
DeepSeek-R1-Distill-Qwen-7B	0.84	0.32	0.50	0.20	0.37	0.40
H2O	0.74	0.26	0.23	0.15	0.18	0.28
RaaS	0.82	0.30	0.40	0.17	0.23	0.32
Quest	**0.86**	0.38	0.43	0.16	0.34	0.34
Arkvale	0.84	0.44	**0.53**	0.20	0.38	0.38
LouisKV	**0.86**	**0.48**	0.50	**0.22**	**0.39**	**0.42**

Table 2: Accuracy comparison on Long-Output Reasoning tasks with a KV cache budget of 1024 tokens across all methods. The best results for each model family (excluding the baseline) are in **bold**.

D.2 Memory Footprint Analysis

In this section, we analyze the KV cache memory footprint of different methods. We assume the number of layers is L, the number of heads is h, the head dimension is d_h, the input sequence length is n, and the output sequence length is m. All KV cache is stored in FP16, leading to a base factor of $4Lhd_h$. As detailed in Table 3, FullCache retains the entire KV cache on the GPU. Quest builds

Method	Memory	Parameters
FullCache	$4Lhd_h(n+m)$	-
Quest	$4Lhd_h(n+m+\frac{n+m}{p})$	page size: p
Arkvale	$4Lhd_h(B+\frac{n+m}{p})$	page size: p, budget size: B
LouisKV	$4Lhd_h(B+\frac{n}{2c}+\frac{m}{2s})$	cluster size: c, segment size: s, budget size: B

Table 3: Theoretical analysis of GPU KV cache memory footprint for different methods, where n is the input length and m is the output length.

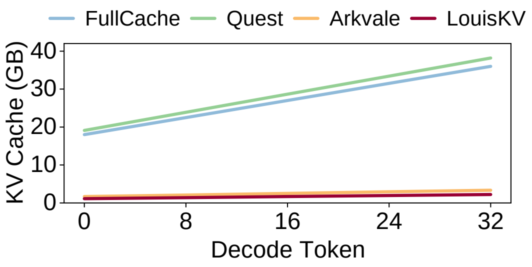


Figure 10: Comparison of KV cache memory consumption during the decoding phase for different methods, with a prefill length of 32K tokens and a batch size of 4.

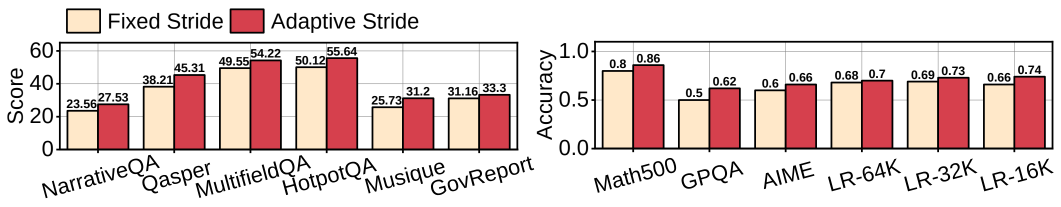


Figure 11: Ablation study on retrieval strategy. We compare the inference accuracy of our adaptive semantic-aware retrieval against a fixed-stride retrieval baseline. The adaptive strategy achieves higher accuracy across most tasks by retrieving information at critical decoding points.

upon this by adding indexing overhead for each page of size p. Both Arkvale and LouisKV are offloading methods that keep a budget of size B on the GPU, but they differ in their indexing overhead. Specifically, both Quest and Arkvale create two index vectors for each key page, whereas LouisKV requires only one for each semantic cluster or segment. We empirically validate this analysis by comparing memory consumption in a long-input scenario (32K prefill tokens), with results shown in Figure 10. The memory usage of FullCache and Quest grows linearly with decoded tokens, making them susceptible to out-of-memory errors. In contrast, Arkvale and LouisKV maintain consistently low memory footprints. As future work, LouisKV's memory consumption could be further reduced by offloading KV cache indices to CPU DRAM.

D.3 Effectiveness of Core Strategies

In this section, we conduct ablation studies to independently analyze the effectiveness of LouisKV's two core designs, *adaptive semantic-aware retrieval strategy* and *decoupled fine-grained management scheme*, to increase inference accuracy. We follow the parameter settings in Section 5, testing on long-input understanding tasks with the Llama3.1-8B-Instruct model and on long-output reasoning tasks with the Qwen3-8B model.

Fixed Stride vs. Adaptive Stride. To validate the superiority of our adaptive retrieval strategy, we compare it against a baseline that employs a fixed-stride retrieval. For a fair comparison, the

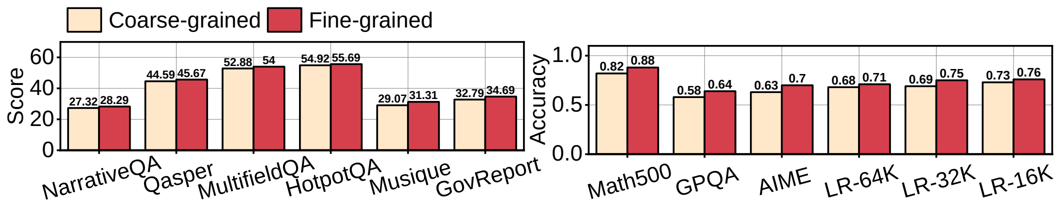


Figure 12: Ablation study on management scheme. We compare the inference accuracy of our decoupled, fine-grained management against a coarse-grained page-level baseline. The superior performance of our fine-grained approach validates the necessity of its tailored design: semantic clusters for inputs and temporal segments for outputs.

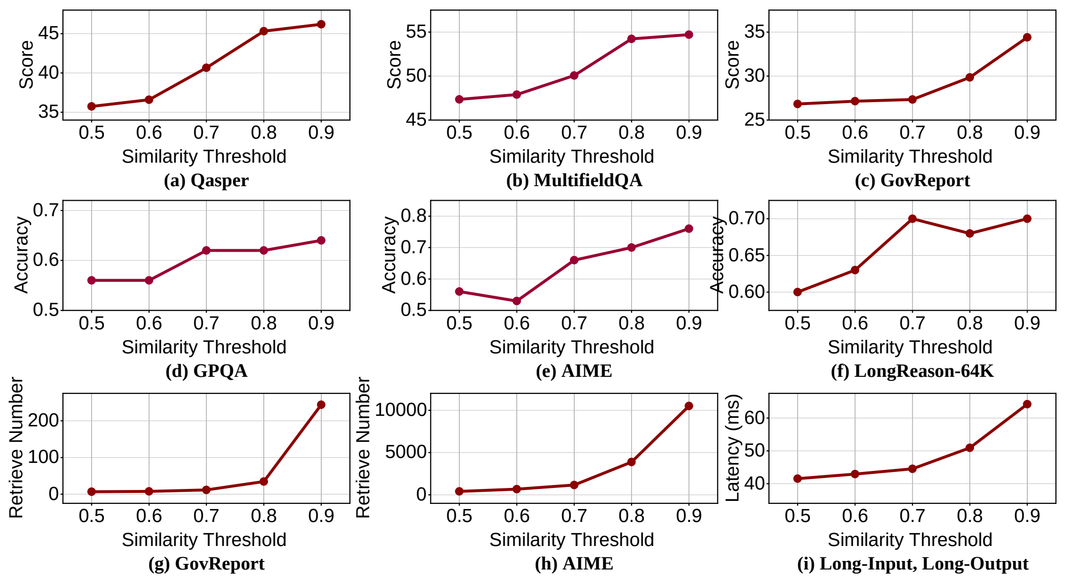


Figure 13: Impact of the similarity threshold τ on inference accuracy, retrieval frequency, and latency. Subplots (a-f) show the accuracy on various long-input and long-output benchmarks. Subplots (g-h) illustrate the corresponding increase in retrieval frequency, and subplot (i) shows the resulting impact on inference latency. The results demonstrate the trade-off between accuracy and efficiency governed by τ.

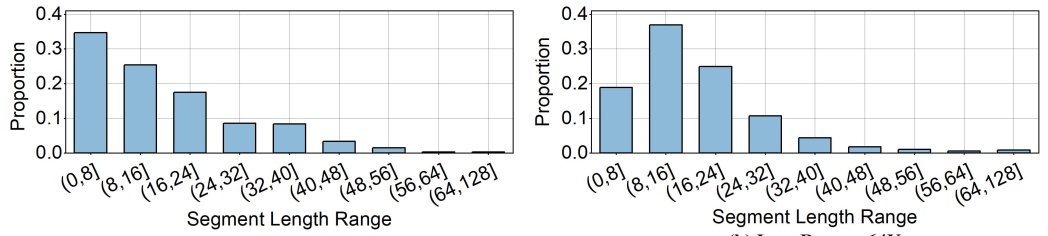
baseline triggers a retrieval every 5 tokens for long-input tasks and every 16 tokens for long-output tasks. These values are chosen to closely match the average retrieval frequency of our adaptive strategy on the respective tasks. As shown in Figure 11(a), our adaptive retrieval strategy achieves higher accuracy across the majority of tasks. This result strongly indicates that a fixed-stride retrieval method can degrade accuracy by failing to retrieve information at critical points. In contrast, our adaptive strategy more precisely identifies the critical decoding points where information must be updated, leading to better inference accuracy.

Coarse-grained vs. Fine-grained Management. Next, we evaluate the contribution of our decoupled, fine-grained management mechanism, comparing it against a baseline that applies a uniform page-level management strategy. To ensure a fair comparison, we force both our strategy and the baseline to perform a retrieval at every decoding step. We set the page size of baseline to 16, aligning it with the average cluster and segment size used in our approach. As depicted in Figure 12(b), our fine-grained management strategy consistently outperforms the coarse-grained baseline. This result confirms the necessity of our tailored design: semantic clusters are better suited for the sparse attention patterns of input sequences, while temporal segments effectively capture the dense attention patterns of output sequences. This tailored design is crucial for achieving high inference accuracy.

D.4 Impact of Similarity Threshold τ.

The threshold τ is a key hyperparameter that governs the trade-off between accuracy and efficiency. It defines the sensitivity for triggering a new KV retrieval operation. A lower value of τ imposes a stricter criterion for identifying a semantic boundary, thereby reducing the frequency of retrievals. While this enhances inference efficiency, it typically leads to accuracy degradation by potentially failing to update the set of critical KVs. Conversely, a higher τ results in better inference accuracy. However, the frequent retrievals incur additional computational and data transfer overhead, thus reducing overall efficiency. We experimentally analyze the impact of τ on both inference accuracy and efficiency.

Analysis of Inference Accuracy. To quantify the impact of τ, we conducted a series of ablation studies on long-input and long-output tasks using the Llama-3.1-8B and Qwen3-8B models, respectively. As shown in Figure 13, a consistent trend is observed: model accuracy improves on all tasks with an increasing τ. This is because a higher τ leads to more frequent retrievals, ensuring that critical information is recalled promptly. More importantly, long-input tasks (Figure 13(a-c)) ex-



(a) MATH500 (b) LongReason-64K

Figure 14: **Distribution of semantic segment lengths.** Histograms for (a) MATH500 and (b) LongReason-64K. The long-tailed distribution indicates that reasoning tasks frequently form long stable semantic segments, which allows LouisKV to effectively amortize the retrieval overhead by skipping redundant operations during these stable periods.

hibit significantly higher sensitivity to the value τ than long-output reasoning tasks (Figure 13(d-f)). We attribute this to the difference in task nature: to generate an answer, long-input understanding tasks often require shifting attention between disparate parts of the context, thus demanding a more responsive $\tan$ to capture these semantic changes. In contrast, long-output reasoning tasks exhibit stronger local coherence, as each reasoning step primarily attends to consistent context. This results in naturally longer semantic segments, making the model's performance less sensitive to the precise value of $\tan$.

Analysis of Inference Efficiency. To quantify the impact of τ on efficiency, we measured the retrieval frequency and inference latency across different thresholds. As shown in Figure 13(g-h), the number of retrievals increases with τ for both task types. Since each retrieval introduces non-negligible overhead, the inference latency consequently increases as well. We measured the average per-token inference latency for different τ values in a long-input, long-output (32K+16K) scenario using Qwen3-8B, as depicted in Figure 13(i). The results show that for Qwen3-8B, $\tau = 0.7$ represents an ideal balance, achieving significant efficiency gains while maximally preserving inference accuracy.

D.5 Distribution of Semantic Segment Lenghts.

To substantiate our amortization claim with hard numbers, we perform a detailed analysis using the Qwen3-8B model with the default threshold $\tau = 0.7$.

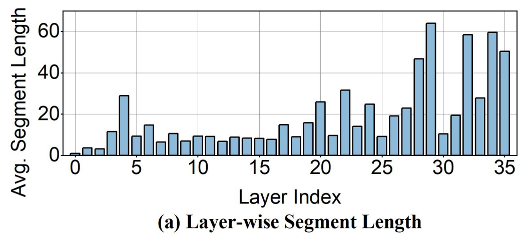
We first measure the average semantic segment length across different tasks. Reasoning tasks like MATH500 and GPQA exhibit an average segment length of approximately 14-18 tokens. This implies that LouisKV reduces the retrieval frequency by more than 10× compared to per-token methods. Even for long-document QA tasks (e.g., NarrativeQA) which involve more frequent topic shifts, the average length remains around 3-6 tokens. This provides strong evidence that LouisKV significantly amortizes the retrieval cost across diverse scenarios.

We further visualize the full distribution of segment lengths in Figure 14 of the paper. The histograms illustrate a long-tailed distribution for reasoning tasks. While short segments are present, a significant proportion of the generation steps fall into long segments (spanning 16 to 128 tokens). These long segments play a crucial role in reducing retrieval overhead, thereby contributing substantially to the overall end-to-end speed improvement.

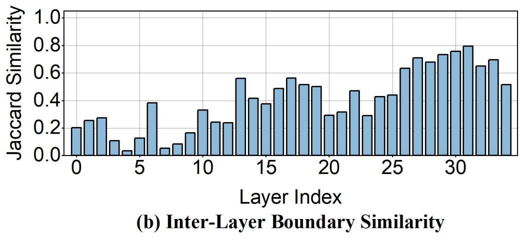
D.6 per-layer boundary analysis

To provide a rigorous justification for our layer-wise retrieval design, we analyze the similarity of semantic boundary detection across different Transformer layers. The analysis is conducted using the Qwen3-8B model on the MATH500 dataset.

Layer-wise Segment Length. We first measure the average length of semantic segments at each layer. As shown in Figure 15(a) of the paper, shallow layers (Layers 0-3) exhibit shorter average



(a) Layer-wise Segment Length



(b) Inter-Layer Boundary Similarity

Figure 15: **Per-layer analysis of semantic segmentation.** (a) Average semantic segment length at each Transformer layer. Shallow layers (e.g., 0-10) exhibit shorter segments while deep layers (e.g., 28-34) form longer segments. (b) Jaccard similarity of boundary trigger indices between adjacent layers. The low similarity across most layers indicates that semantic boundaries are detected at different points, validating the necessity of our per-layer retrieval strategy.

segments (approximately 4-6 tokens). This observation aligns with the intuition that lower layers capture lower-level, rapidly changing features such as syntactic structures or lexical choices, thus identifying more frequent semantic boundaries. In stark contrast, deep layers (e.g., Layers 28-34) exhibit substantially longer segments. This reflects their function in processing high-level abstract concepts, which remain stable during extended reasoning chains.

Inter-Layer Similarity on Boundaries. To further quantify the similarity in trigger points across layers, we calculate the Jaccard similarity of the boundary-triggering token indices between adjacent layers. As illustrated in Figure 15(b), the similarity is notably low, especially in the shallow layers. This low similarity indicates that a semantic shift occurring in a lower layer does not necessarily result in a corresponding shift in a higher layer. This phenomenon validates the necessity of our per-layer segmentation strategy, which enables each layer to independently manage the retrieval and segmentation frequencies.

E USE OF LLMS

In this research, we utilized Large Language Models (LLMs) as an auxiliary tool to enhance research efficiency and the quality of the manuscript. The authors maintained strict oversight throughout this process, ensuring adherence to the principles of academic integrity. We affirm that the authors bear full and final responsibility for all content, analyses, and conclusions presented herein. A detailed breakdown of our specific use cases for these models is provided below:

LANGUAGE POLISHING AND COPY-EDITING

We utilized LLMs to assist with language refinement in this manuscript. Its use was confined to improving grammar, clarity, and overall readability. The authors reviewed all suggested changes to ensure the integrity of our original arguments and that the scientific content remained unaltered.

LITERATURE REVIEW AND BRAINSTORMING

In the preliminary stages of research, an LLM was used to summarize established concepts and survey related work, which helped in identifying potential research gaps. However, all final literature citations and core research ideas were independently developed, verified, and validated by the authors.