KVSCOUT: EFFICIENT LONG-CONTEXT DECODING WITH SENTENCE-LEVEL KV RETRIEVAL

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-context models show strong capabilities in processing long inputs but incur substantial memory and computational costs due to the growing KV cache. KV retrieval methods reduce these costs by keeping the full context KV cache in CPU memory and retrieving only a small subset to the GPU for attention. However, frequent KV selection and transfer can offset sparse-attention savings and limit decoding speedup, motivating fewer retrievals during decoding. Yet, existing selectors are designed for next-token retrieval, and reusing their selections across more decoding steps degrades generation quality. To address these limitations, we propose **KVScout**, an efficient KV retrieval method that learns to select the KV entries needed for generating the upcoming sentence. With a lightweight retriever (8.9M parameters), KVScout builds a contextualized sentence index during prefill and queries it with the generation state before generating each sentence, enabling each retrieved KV window to be reused throughout that sentence. Extensive experiments on three long-context benchmarks demonstrate that KVScout maintains generation quality with a low refresh frequency, achieving $5.3\times$ and 1.7–$3.7\times$ higher throughput than full-cache decoding and existing KV retrieval methods at 256K context, respectively.

1 INTRODUCTION

Long-context models (OpenAI, 2026b; Yang et al., 2025a; Meta, 2024) support diverse tasks that require processing and integrating information across extensive inputs, including multi-document question answering (Yang et al., 2018), code generation (Zhang et al., 2023a), and document summarization (Huang et al., 2021). However, efficient decoding becomes increasingly challenging as the context grows (Kwon et al., 2023): the key–value (KV) cache consumes more GPU memory, and attending to this cache requires more computation and memory access at each generation step. KV retrieval methods (Tang et al., 2024; Lee et al., 2025) reduce attention cost by selecting a small subset of context KV entries during decoding. By offloading the full KV cache to CPU memory and loading only the selected entries (Sun et al., 2025; Yang et al., 2025c), these methods also reduce GPU memory usage.

However, KV selection and CPU-to-GPU transfer introduce additional decoding costs for these KV retrieval methods (Yang et al., 2025b; Liu et al., 2026). For example, at a 64K context length, these operations increase the profiled CPU time by more than 40% for Quest (Tang et al., 2024) and ShadowKV (Sun et al., 2025), even with a KV budget of only 512 tokens (as shown in Section 3.2). To reduce these costs, methods such as InfiniteHiP (Lee et al., 2025) perform KV selection less frequently and reuse the selected subsets across multiple decoding steps. Although more efficient, these methods still select KV subsets primarily to match the next-token attention distribution at selection time. These subsets therefore struggle to cover the context needed by subsequent tokens as generation proceeds, and generation quality degrades substantially at longer refresh intervals (Section 3.3). This motivates a selection method that anticipates the context required across multiple upcoming decoding steps, enabling less frequent retrieval while preserving generation quality.

To address this challenge, we introduce **KVScout**, a sentence-level predictive KV retrieval method that learns to *scout* context KV entries for the upcoming sentence. During prefill, we use a learnable special-token probe to build a compact, contextualized sentence index without altering the original context KV states, and offload the complete context cache to CPU memory. At each sentence bound-

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.

ary during decoding, KVScout uses another learnable special-token probe conditioned on the current generation prefix and visible context to generate a retrieval query for querying the sentence index and retrieving candidate context sentences relevant to the upcoming sentence. We then fetch the KV entries corresponding to these candidates and apply token-level refinement to select the entries that complement a persistent KV set within the context KV budget. Unlike existing methods that refresh the window at nearly every token, KVScout reuses the selected window across multiple decoding steps in one sentence, significantly amortizing KV selection and transfer costs. With the base model frozen, we train only the probe embeddings and lightweight projection modules (0.11% of Qwen3-8B's parameter size) using a query–sentence contrastive objective to align retrieval queries with contextualized sentence indexes relevant to the upcoming sentence.

We evaluate KVScout on three long-context benchmarks: LongBench (Bai et al., 2024), RULER (Hsieh et al., 2024), and Needle-in-a-Haystack (Kamradt, 2023). Results across different models, context lengths, and KV budgets show that KVScout can effectively retrieve relevant context KV entries and consistently achieve the highest average generation quality among existing KV selection methods. Beyond quality, KVScout effectively amortizes KV selection and transfer costs through multi-step window reuse, achieving a $5.3\times$ throughput speedup over full-cache decoding at 256K context. Under batched decoding, KVScout can achieve up to an $8.8\times$ throughput speedup over full-cache decoding and a $1.8\times$ speedup over ShadowKV on one NVIDIA H20 GPU.

Our contributions are summarized as follows: (i) We show that frequent KV reselection incurs substantial selection and transfer overhead, significantly limiting the decoding speedup of dynamic KV retrieval, yet reducing the reselection frequency without compromising generation quality remains challenging. (ii) We propose KVScout, a sentence-level predictive KV retrieval method that anticipates the context needed for the upcoming sentence and reuses the retrieved KV entries throughout its generation, shifting KV retrieval from reacting to the current token to proactively serving multiple future decoding steps. (iii) We demonstrate that KVScout achieves the highest average generation quality among existing KV selection methods across models and KV-cache budgets while delivering substantial decoding speedups.

2 Related Work

KV cache eviction reduces GPU memory by retaining a limited subset of context tokens in the KV cache and permanently discarding the rest. H2O (Zhang et al., 2023b) and TOVA (Oren et al., 2024) select tokens using accumulated and current attention scores, respectively, and update the retained cache during decoding. StreamingLLM (Xiao et al., 2024b) preserves attention sinks and a recent token window to support streaming generation. SnapKV (Li et al., 2024) selects important context tokens using an observation window at the end of the prompt. PyramidKV (Cai et al., 2025b) and Ada-KV (Feng et al., 2025) further adapt cache budgets across layers and attention heads, respectively. R-KV (Cai et al., 2025a) and ThinKV (Ramachandran et al., 2026) further focus on evicting the KV cache of generated tokens. However, discarded entries cannot be recovered without recomputation. As the context needed by generation shifts across reasoning steps, evidence evicted early becomes inaccessible when it is later required.

KV cache retrieval avoids this loss by keeping the complete context KV cache, typically offloaded to CPU memory, and loading a query-dependent subset to the GPU during decoding. To select KV entries, Quest (Tang et al., 2024) bounds each page's attention score using its elementwise minimum and maximum keys, and ArkVale (Chen et al., 2024) summarizes each page into a digest to estimate its importance. InfLLM (Xiao et al., 2024a) and ShadowKV (Sun et al., 2025) score blocks and chunks through representative keys, respectively. SentenceKV (Zhu et al., 2025) averages keys within each sentence into a semantic vector and selects sentences by their similarity to the query. AttentionPredictor (Yang et al., 2025c) learns to predict next-step attention scores from the attention history. These methods reselect KV entries at nearly every decoding step, repeatedly incurring selection and transfer costs. TidalDecode (Yang et al., 2025b) and OmniKV (Hao et al., 2025) reduce these costs by sharing selections across layers, and InfiniteHiP (Lee et al., 2025) and TokenSelect (Wu et al., 2025) reuse selections across decoding steps. However, these methods select KV entries for the current decoding step, which cannot adequately support multiple subsequent steps (as shown in Section 3.3). KVScout instead learns to select the context needed by the entire upcoming sentence, allowing each retrieved window to be efficiently reused throughout its generation.

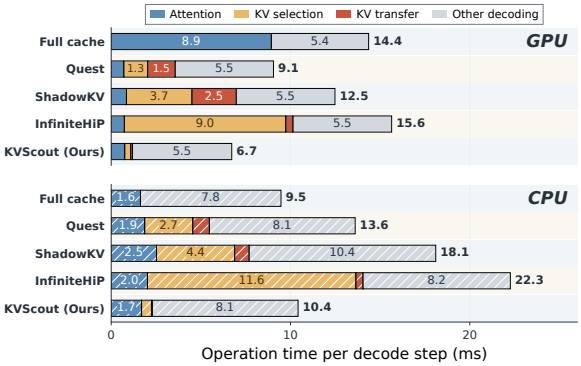


Figure 1: CPU and GPU time breakdown at 64K context.

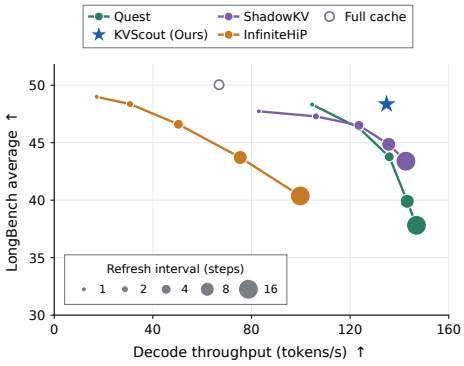


Figure 2: Throughput–quality tradeoff. Circle size indicates refresh interval s.

3 Rethinking KV Retrieval for Efficient Decoding

In this section, we analyze the limitations of existing KV retrieval methods for long-context decoding. We first show in Section 3.2 that frequent KV retrieval incurs substantial selection and transfer overhead, which can offset the savings from sparse attention. Reusing the selected KV window across more decoding steps can amortize this overhead, but Section 3.3 shows that naively extending the reuse interval degrades generation quality. These observations motivate a selection strategy that anticipates the context needed by upcoming decoding steps, allowing each retrieved KV window to remain useful throughout a longer reuse interval.

3.1 Preliminaries

Let $t \in \{1, \ldots, T\}$ denote the decoding step, and let $\mathcal{C}$ be the context KV cache produced during prefill. We consider an offloaded retrieval setting in which $\mathcal{C}$ resides in CPU memory, while only a subset $\mathcal{W}_t \subseteq \mathcal{C}$ is available on the GPU for context attention at step t. A retrieval rule g determines the contents of $\mathcal{W}_t$ and the decoding steps at which it is refreshed. Each KV retrieval consists of two operations: (i) selecting the entries needed for the subsequent tokens, and (ii) fetching those not already resident on the GPU. Let c_{att} denote the cost of attending to one entry, c_{sel} the cost of identifying the relevant entries at one update, and c_{tr} the cost of transferring one byte. If each cache entry occupies κ bytes, the cost of decoding T tokens under g is then

$$\text{Cost}(g) = c_{\text{att}} \sum_{t=1}^{T} |\mathcal{W}_t| + \sum_{t \in \mathcal{T}} (c_{\text{sel}} + c_{\text{tr}}\, \Delta_t), \qquad \Delta_t = \kappa\, |\mathcal{W}_t \setminus \mathcal{W}_{t-1}|, \tag{1}$$

where $|\mathcal{W}_t|$ is the number of context KV entries in the GPU-resident window at step t, $\mathcal{T}$ is the set of steps at which g updates the window, and Δ_t is the number of bytes transferred at step t. The first term captures the *attention cost* across decoding steps, while the second captures the *KV retrieval cost*, including KV selection and transfer at each retrieval.

3.2 Observation 1: KV Retrieval Overhead Limits Decoding Speedup

For cache-retrieval methods, a fixed KV budget bounds the attention cost in Equation (1), but retrieving KV entries incurs additional selection and transfer costs. Whether a smaller attention window translates into faster decoding therefore depends on the KV retrieval overhead. To examine this tradeoff, we compare full-cache decoding with three representative KV retrieval methods: Quest (Tang et al., 2024), ShadowKV (Sun et al., 2025), and InfiniteHiP (Lee et al., 2025). For a fair comparison, we port all methods into the same inference engine built on the official ShadowKV (Sun et al., 2025) codebase and evaluate them using Qwen3-8B in FP16 on a single NVIDIA H20 GPU with 16 CPU threads. All methods, including full-cache decoding, share the same generic optimizations and compute attention with the FlashAttention-2 kernel (Dao, 2024). We profile CPU and GPU operation times over 384 decoding steps on 64K contexts constructed from extended HotpotQA (Yang et al., 2018), with a KV budget of $B = 512$.

Figure 1 breaks down the CPU and GPU time per decoding step into KV selection, KV transfer, and other decoding operations. As shown, full-cache decoding spends 8.9ms per step on attention, accounting for 62% of its GPU time and making attention the dominant bottleneck. Cache-retrieval methods reduce the attention time substantially by attending to only a small subset of KV entries, with Quest and ShadowKV requiring 0.70 ms and 0.85 ms per step, respectively. However, they perform KV retrieval at every decoding step, repeatedly incurring selection and transfer overheads, with the profiled CPU time increasing from 9.5 ms for full-cache decoding to 13.6 ms (+44%) and 18.1 ms (+91%) for Quest and ShadowKV, respectively. InfiniteHiP amortizes retrieval by refreshing its hierarchical pruning results every four steps, but still incurs 9.0 ms of GPU KV selection and 22.3 ms of CPU time per step. Taken together, these results show that **existing cache-retrieval methods still incur substantial KV retrieval costs**. When the savings in attention computation are limited, particularly at shorter context lengths (e.g., less than 64K in our setting), this cost can outweigh those savings and make retrieval-based decoding slower than full-cache decoding.

3.3 Observation 2: Naively Reducing the Refresh Frequency Degrades Quality

The substantial KV retrieval overhead observed in Section 3.2 motivates reducing the refresh frequency to amortize selection and transfer costs over more decoding steps. With fewer refreshes, each selected KV window must remain useful across multiple decoding steps under the same KV budget. However, existing selectors estimate KV importance from the current decoding query and do not explicitly account for the context needed by subsequent steps within the reuse interval.

To examine whether these selectors can directly support multi-step reuse, we evaluate Quest, ShadowKV and InfiniteHiP with fixed refresh intervals of 1, 2, 4, 8, and 16 decoding steps under the same KV budget ($B = 512$). Figure 2 compares the average quality on 16 LongBench tasks and the decoding throughput at 64K context (under the setting of Section 3.2) across different refresh intervals. As can be seen, **increasing the refresh interval improves decoding throughput but reduces generation quality for existing KV retrieval methods**. Together with the retrieval overhead in Figure 1, these findings motivate selecting KV entries for their usefulness across multiple upcoming steps, to amortize retrieval costs while maintaining generation quality.

4 KVScout

In this section, we introduce KVScout, an efficient KV cache retrieval method that learns to select KV entries needed to generate the upcoming sentence. Motivated by Observation 2 in Section 3.3, we aim to reduce the number of retrievals $|\mathcal{T}|$ by selecting KV subsets that support multiple consecutive decoding steps. A sentence typically expresses a coherent statement or reasoning step (Bogdan et al., 2025), making it a natural unit for considering the context needs of these steps jointly. We therefore select KV entries once at each sentence boundary during generation to serve the entire upcoming sentence. Reusing this window across all of the sentence's decoding steps amortizes the retrieval cost. Section 4.1 describes how KVScout constructs a sentence-level retrieval index during prefill and updates the GPU-resident window at sentence boundaries during decoding. Section 4.2 presents how the predictive query is learned from annotated reasoning traces.

4.1 Sentence-Level Predictive KV Retrieval

To identify the required context before that sentence is generated, we build the contextualized sentence indices during prefilling and use them for predictive sentence retrieval decoding.

Prefilling: Contextualized sentence indexing. To support retrieval from the offloaded context cache, we construct sentence-level embeddings that serve as its retrieval index. Specifically, during prefill, we segment the input tokens $\boldsymbol{x}$ into N sentences $s_1, \ldots, s_N$, and insert a learnable `[SUM]` probe immediately after each sentence. The i-th probe, denoted $\texttt{[SUM]}_i$, attends only to the original input tokens in the prefix ending at s_i. At a fixed intermediate layer ℓ, we extract its hidden state $\boldsymbol{h}^{(\ell)}_{\texttt{[SUM]}_i}$ and project it into a retrieval index embedding:

$$\boldsymbol{e}_i = f_e\big(\boldsymbol{h}^{(\ell)}_{\texttt{[SUM]}_i}\big) \in \mathbb{R}^{d_r}, \tag{2}$$

where f_e is a learned projection and d_r is the embedding dimension. We use $\boldsymbol{e}_i$ as a contextualized retrieval key for sentence s_i and associate it with the KV entries of that sentence's original tokens.

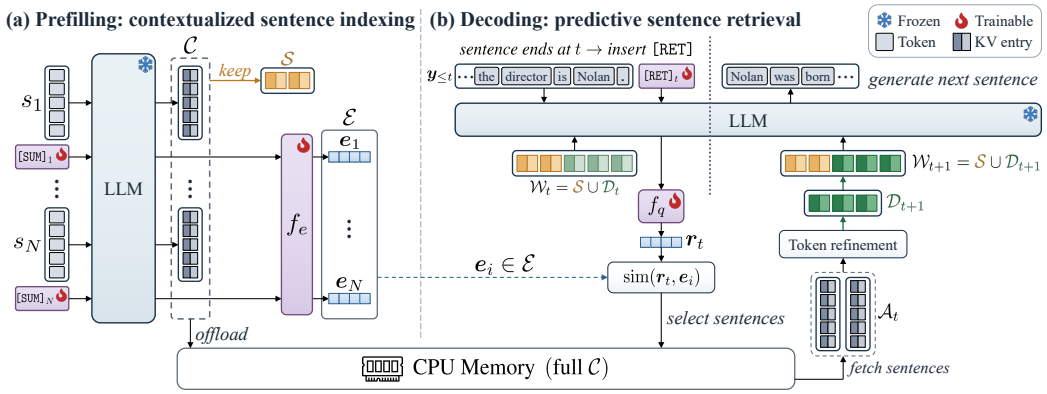


Figure 3: Overview of KVScout. **(a) Prefilling:** each sentence is indexed by the contextualized embedding of its `[SUM]` token, and the full KV cache is offloaded to the CPU. **(b) Decoding:** at each sentence end, a `[RET]` token retrieves relevant sentences by similarity to their `[SUM]` embeddings and fetches their KV entries for decoding the next sentence.

The original input tokens do not attend to any `[SUM]` probe. They retain their standard position indices and causal attention pattern, and so their cached keys and values remain unchanged. A single prefill pass thus produces both the context KV cache $\mathcal{C}$ and the sentence index $\mathcal{E} = \{\boldsymbol{e}_i\}_{i=1}^{N}$. After prefill, $\mathcal{C}$ is offloaded to CPU memory, while $\mathcal{E}$ remains on the GPU for retrieval at generated-sentence boundaries during decoding.

Decoding: Predictive sentence retrieval. Before decoding, we first construct a persistent set $\mathcal{S}$ of context KV entries that remains on the GPU throughout generation. SnapKV (Li et al., 2024) observes that tokens receiving high attention at the end of the input tend to remain important during generation. Motivated by this observation, we aggregate the attention each entry receives from the last m tokens of $\boldsymbol{x}$ and retain the highest-scoring B_s entries as $\mathcal{S}$. The initial GPU-resident window $\mathcal{W}_0$ is formed by the top-B entries under the same ranking.

In addition to these persistent KV entries, the next sentence may require context specific to the current reasoning state. We therefore refresh a dynamic set $\mathcal{D}_t$ of context KV entries at each generated-sentence boundary based on the current generation prefix. The GPU-resident window at step t is thus $\mathcal{W}_t = \mathcal{S} \cup \mathcal{D}_t$, whose size satisfies $|\mathcal{W}_t| \le B$. To retrieve this additional context, KVScout constructs a retrieval query using a temporary learnable probe `[RET]`. Specifically, when a generated token y_t completes one sentence, we insert a special token `[RET]` at position $t+1$. The probe attends to the current reasoning prefix and current window $\mathcal{W}_t$. Its hidden state $\boldsymbol{h}^{(\ell)}_{\texttt{[RET]}_t}$ at layer ℓ is projected into the following retrieval query embedding:

$$\boldsymbol{r}_t = f_q\big(\boldsymbol{h}^{(\ell)}_{\texttt{[RET]}_t}\big) \in \mathbb{R}^{d_r}, \tag{3}$$

where f_q is a learned projection. We then score the N sentence embeddings in the index set $\mathcal{E}$ by normalizing the cosine similarities between $\boldsymbol{r}_t$ and these sentence embeddings:

$$\pi_{t,i} = \frac{\exp\big(\mathrm{sim}(\boldsymbol{r}_t, \boldsymbol{e}_i)/\tau\big)}{\sum_{j=1}^{N} \exp\big(\mathrm{sim}(\boldsymbol{r}_t, \boldsymbol{e}_j)/\tau\big)}, \quad i \in \{1, \dots, N\}, \tag{4}$$

where $\mathrm{sim}(\cdot,\cdot)$ denotes the cosine similarity, and τ is a temperature parameter. We refer to the learnable `[SUM]` and `[RET]` probes, together with the projections f_e and f_q, as our *retriever*.

The number of context sentences needed to support the next sentence varies across reasoning steps. For example, a step grounded in a specific fact may rely on a single context sentence, whereas a synthesis step may draw on several. We therefore determine the number of retrieved sentences adaptively and select the smallest set of top-ranked sentences whose cumulative probability mass reaches a threshold $p \in (0,1)$. Specifically, let $i_{(1)}, \dots, i_{(N)}$ be the sentence indices sorted such that $\pi_{t,i_{(1)}} \ge \cdots \ge \pi_{t,i_{(N)}}$, the retrieved set $\mathcal{A}_t$ is

$$\mathcal{A}_t = \{s_{i_{(1)}}, \dots, s_{i_{(n_t^\star)}}\}, \quad n_t^\star = \min\Big\{n : \sum_{k=1}^{n} \pi_{t,i_{(k)}} \ge p\Big\}. \tag{5}$$

Because the retrieved sentences vary in length, their KV entries may exceed or fall short of the dynamic-set budget B_d. If the retrieved sentences provide too few KV entries to fill the dynamic set, i.e., $\sum_{s \in \mathcal{A}_t} |s| < B_d$, we extend $\mathcal{A}_t$ with additional sentences in descending order of $\pi_{t,i}$ until the resulting KV-entry pool is large enough to cover the dynamic-set budget. We then fetch the retrieved KV entries from CPU memory and refine this pool at the token level. Recent works (Yang et al., 2025c; Li et al., 2024) show that a high similarity between consecutive queries leads to stable attention scores over cached tokens, motivating us to use recent queries for token selection. Specifically, we use the queries from the most recent w decoding steps as an observation window and aggregate their attention scores over the retrieved KV entries. The top-B_d entries are retained as $\mathcal{D}_{t+1}$, yielding the updated GPU-resident window $\mathcal{W}_{t+1} = \mathcal{S} \cup \mathcal{D}_{t+1}$, which is reused throughout the next sentence. Algorithm 1 in Appendix A summarizes the complete inference procedure.

4.2 Training

Constructing sentence-level supervision. Training our retriever requires annotations identifying which context sentences are important for generating the next sentence. To obtain this supervision, we first construct long-context QA samples from the HotpotQA (Yang et al., 2018) training split following the procedure of LongBench (Bai et al., 2024). For each question, we include the full Wikipedia[1] articles containing its supporting paragraphs, add distractor articles, and randomly order the selected articles to form the input context.

To turn these QA samples into sentence-level retrieval supervision, we generate reasoning traces with Qwen3-8B (Yang et al., 2025a) under the full KV cache. We split each trace into sentences and treat each sentence as a *target sentence*. For each target sentence, we use `gpt-5.6-luna` (OpenAI, 2026a) to identify the context sentences relevant to generating it. Each training sample then pairs the generation prefix preceding a target sentence with the indices of its relevant context sentences, providing supervision for retrieval before the target sentence is generated. This yields approximately 170k sentence-level annotations across 9,879 QA samples for training.

Training objective. Consider a training sample whose target sentence begins at sentence boundary t. We compute the context sentence embeddings $\boldsymbol{e}_i$ and the retrieval query $\boldsymbol{r}_t$ as in Equation (2) and Equation (3), respectively, where the `[RET]` probe attends to the full input context and the generation prefix up to boundary t. Let $\mathcal{P}_t$ denote the indices of context sentences annotated as relevant to the target sentence. We randomly sample a negative pool $\mathcal{N}$ from sentences in the same input context that are not annotated as relevant to any sentence in the reasoning trace. Our objective is to increase the retrieval query's similarity to all positive context sentences and decrease its similarity to all sampled negatives. We therefore use a query–sentence contrastive loss:

$$\mathcal{L}_t = -\frac{1}{|\mathcal{P}_t|} \sum_{i \in \mathcal{P}_t} \log \frac{\exp\big(\mathrm{sim}(\boldsymbol{r}_t, \boldsymbol{e}_i)/\tau\big)}{\exp\big(\mathrm{sim}(\boldsymbol{r}_t, \boldsymbol{e}_i)/\tau\big) + \sum_{j \in \mathcal{N}} \exp\big(\mathrm{sim}(\boldsymbol{r}_t, \boldsymbol{e}_j)/\tau\big)}. \tag{6}$$

We minimize this loss by updating only the `[SUM]` and `[RET]` input embeddings and the projection heads f_e and f_q, while keeping the base model frozen. Training is thus parameter-efficient, with only 8.92M trainable parameters for Qwen3-8B, less than 0.11% of the base model's parameters.

5 Experiments

5.1 Experimental Setup

Benchmarks and Models. To evaluate KVScout's performance on long-context inference, we use LongBench (Bai et al., 2024), which includes 16 tasks spanning six categories: single- and multi-document QA (SQA and MQA), summarization, few-shot learning, synthetic tasks, and code completion.[2] Following Yang et al. (2025c), we further evaluate on RULER QA (Hsieh et al., 2024) and the Needle-in-a-Haystack (NIAH) (Kamradt, 2023) benchmarks, which assess long-context question answering and information retrieval, respectively. We conduct experiments on Qwen3-8B (Yang et al., 2025a) and DeepSeek-R1-Distill-Llama-8B (Guo et al., 2025), both with thinking enabled.

[1] https://dumps.wikimedia.org/enwiki/

[2] Our training data are built from the HotpotQA training split (Section 4.2), disjoint from the validation split used by LongBench.

Table 1: LongBench results on Qwen3-8B and DeepSeek-R1-Distill-Llama-8B at three KV cache budgets. **Bold** indicates the best within each budget, and underlining indicates the second-best.

Budget	Method	SQA	MQA	Summary	Few-shot	Synthetic	Code	Average↑	SQA	MQA	Summary	Few-shot	Synthetic	Code	Average↑
		Qwen3-8B							*DeepSeek-R1-Distill-Llama-8B*						
Full	Full cache	40.62	64.46	23.49	64.18	64.36	46.85	50.04	31.88	47.20	23.92	59.65	56.26	30.21	41.31
512	SnapKV	38.55	49.60	20.64	61.39	53.34	36.87	43.19	28.62	37.52	20.96	56.63	43.80	28.17	35.95
	Ada-KV	38.24	49.13	20.51	62.29	51.56	36.30	42.89	29.31	37.08	21.35	56.64	39.77	26.34	35.33
	Quest	39.30	62.83	22.98	62.66	**59.60**	**45.28**	48.32	31.02	45.25	23.17	56.76	44.69	29.70	38.59
	AttentionPredictor	41.00	61.46	22.61	62.50	53.66	42.90	47.24	30.69	**49.80**	23.51	56.17	37.61	27.58	38.18
	ShadowKV	**41.14**	61.46	**24.00**	63.93	52.50	43.53	47.73	**31.27**	47.00	**23.68**	**58.27**	44.68	27.68	39.09
	SentenceKV	38.75	55.94	22.36	63.34	56.05	41.27	45.99	30.96	42.25	22.78	58.08	41.26	28.94	37.66
	InfiniteHiP	39.21	59.57	22.34	61.31	54.15	44.99	46.60	30.18	46.75	22.85	56.41	34.68	30.29	37.41
	KVScout	40.51	**63.17**	22.56	**64.07**	57.83	43.49	**48.35**	30.26	47.46	22.80	57.10	**47.55**	**30.39**	**39.30**
1024	SnapKV	40.00	58.04	22.03	63.76	57.82	41.61	46.90	30.69	42.14	22.59	58.95	50.38	28.80	38.84
	Ada-KV	40.56	56.78	22.14	63.91	56.52	39.97	46.45	30.50	42.80	22.55	58.61	49.12	28.86	38.71
	Quest	40.78	62.38	23.27	64.54	**62.28**	**47.35**	49.51	32.80	46.06	23.45	58.18	**53.58**	30.07	40.55
	AttentionPredictor	40.07	64.73	23.15	64.54	58.95	46.16	49.23	32.66	**49.66**	23.60	57.68	49.32	30.08	40.60
	ShadowKV	40.24	62.12	**23.98**	64.85	56.75	44.54	48.51	30.34	48.68	**23.65**	58.82	49.76	27.57	39.95
	SentenceKV	**41.68**	60.95	23.31	63.76	60.43	44.08	48.63	31.24	47.26	23.51	58.98	50.38	29.85	40.22
	InfiniteHiP	40.42	62.67	23.03	64.35	59.30	45.75	48.84	31.45	47.16	23.33	**59.24**	48.43	**31.64**	40.23
	KVScout	40.90	**64.80**	23.17	**64.97**	61.05	45.00	**49.60**	**33.00**	48.62	23.60	58.77	51.67	30.23	**40.99**
2048	SnapKV	41.00	62.07	23.03	63.40	60.75	44.20	48.65	31.06	44.82	23.28	59.83	53.30	30.20	40.25
	Ada-KV	41.44	61.69	22.90	63.89	63.52	44.19	49.07	30.75	45.80	23.41	60.47	50.62	30.56	40.23
	Quest	**41.67**	63.33	23.44	63.99	63.75	46.91	49.91	32.01	47.82	23.54	59.61	55.13	30.10	41.21
	AttentionPredictor	41.39	**64.95**	23.20	65.06	62.39	46.55	50.11	32.22	47.54	23.47	**61.01**	54.17	**31.19**	41.46
	ShadowKV	40.93	61.86	**23.91**	65.15	56.75	43.71	48.53	31.36	45.35	**23.95**	59.62	52.36	28.31	40.14
	SentenceKV	41.10	63.38	23.37	64.98	63.05	44.88	49.65	31.26	47.16	23.83	60.81	52.90	31.01	41.06
	InfiniteHiP	41.01	63.09	23.16	64.67	62.53	46.23	49.58	32.06	47.49	23.73	58.81	52.33	29.71	40.65
	KVScout	41.07	64.87	23.33	**65.37**	**64.00**	**47.10**	**50.38**	**32.45**	**48.50**	23.84	59.61	**56.76**	30.53	**41.74**

Baselines. We compare KVScout with full-cache decoding and seven state-of-the-art KV selection baselines, grouped into the two categories in Section 2: (i) KV cache eviction methods: *SnapKV* (Li et al., 2024) and *Ada-KV* (Feng et al., 2025); and (ii) KV cache retrieval methods: *Quest* (Tang et al., 2024), *AttentionPredictor* (Yang et al., 2025c), *ShadowKV* (Sun et al., 2025), *SentenceKV* (Zhu et al., 2025), and *InfiniteHiP* (Lee et al., 2025). We additionally report *Full cache* decoding as an upper-bound reference. Both the evaluated Qwen3-8B and DeepSeek-R1-Distill-Llama-8B use grouped-query attention (GQA). To ensure a fair comparison under the same KV budget, we follow Feng et al. (2025) and average selection scores across query heads that share a KV head for SnapKV, Ada-KV, Quest, SentenceKV, InfiniteHiP, and KVScout.

Implementation details. At inference, we use greedy decoding across all methods and evaluate KV-cache budgets B of 512, 1024, and 2048 tokens. For both models, we set the size of $\mathcal{D}_t$ to 192 for the 512-token budget and 640 for the 1024- and 2048-token budgets. We use the final $m = 32$ input tokens to determine the fixed set $\mathcal{S}$. During decoding, we retrieve a top-p candidate pool with $p = 0.99$, using the same temperature $\tau = 0.05$ as in training. We refine the candidate pool using queries from the $w = 32$ most recently generated tokens. Additional implementation details, including retriever architecture, training configuration, and the sentence-boundary detection rule, are provided in Appendix C. All experiments are conducted on a single NVIDIA H20 GPU.

5.2 Generation Quality

LongBench. Table 1 summarizes the LongBench results by task group for Qwen3-8B and DeepSeek-R1-Distill-Llama-8B under three KV budgets, with overall scores averaged over 16 tasks. The detailed per-task results on Qwen3-8B and DeepSeek-R1-Distill-Llama-8B are reported in Tables 7 and 8 in Appendix D.4, respectively. As can be seen, KVScout achieves the highest average score among compressed methods across all six model–budget combinations. At $B = 2048$, it exceeds full-cache decoding by 0.34 points on Qwen3-8B and 0.43 points on DeepSeek-R1, showing that it can preserve task quality under limited KV budgets. KVScout achieves higher average scores than the eviction methods SnapKV and Ada-KV, showing that recalling relevant context during generation preserves task quality more effectively than evicting it. KVScout also outperforms the retrieval methods Quest, AttentionPredictor, ShadowKV, SentenceKV, and InfiniteHiP, supporting the effectiveness of its predictive sentence-level retrieval (see case studies in Appendix B).

RULER QA. Table 2 presents the RULER QA results on Qwen3-8B with $B = 2048$ at context lengths of 4K, 16K, and 28K tokens. Consistent with the LongBench results, KVScout achieves the highest average accuracy and exceeds full-cache decoding on every task and context length. At 28K,

Table 2: RULER QA accuracy (%) on Qwen3-8B with a budget of $B = 2048$. **Bold** indicates the best performance.

Method	QA1			QA2			Avg.↑
	4K	16K	28K	4K	16K	28K	
Full cache	86.7	78.0	80.0	71.3	70.7	66.7	75.6
SnapKV	86.0	74.7	74.7	70.7	63.3	57.3	71.1
Ada-KV	86.7	78.7	69.3	70.7	62.0	58.7	71.0
Quest	84.0	78.0	68.0	71.3	69.3	65.3	72.7
AttentionPredictor	85.3	76.0	78.7	72.0	68.0	**68.0**	74.7
ShadowKV	86.7	78.7	73.3	71.3	66.7	65.3	73.7
SentenceKV	85.3	73.3	73.3	**72.7**	67.3	62.7	72.4
InfiniteHiP	86.7	71.3	60.0	71.3	64.0	56.7	68.3
KVScout	**87.3**	**79.3**	**81.3**	72.0	**72.0**	**68.0**	**76.7**

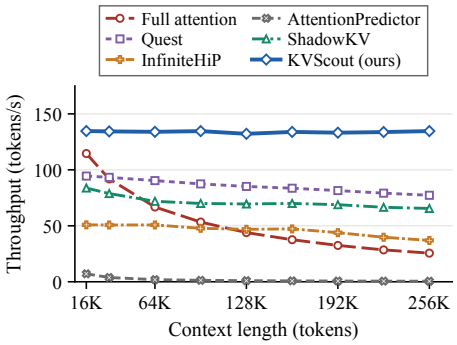


Figure 4: Decoding throughput versus context length on Qwen3-8B ($B = 1024$).

KVScout maintains the highest accuracy while most baselines fall well below full-cache decoding, showing that sentence-level predictive retrieval remains effective as the context grows.

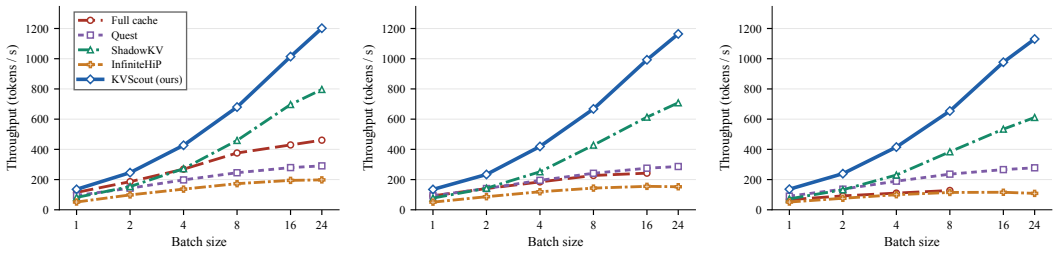
Needle-in-a-Haystack. We evaluate KVScout on the Needle-in-a-Haystack (NIAH) test across six context lengths and 11 insertion depths. Figure 12 in Appendix D.5 reports the retrieval accuracy of KVScout, Quest, and the full-cache baseline on Qwen3-8B and DeepSeek-R1-Distill-Llama-8B with KV budgets of $B \in \{1024, 2048\}$. As shown, KVScout outperforms Quest in all settings and achieves accuracy comparable to the full-cache baseline. These results demonstrate the effectiveness of KVScout in preserving critical information for accurate retrieval.

5.3 Efficiency Analysis

Decoding time breakdown. We revisit the 64K profiling setting of Section 3.2 to examine the KV retrieval overhead of KVScout. As shown in Figure 1, KVScout spends only 0.6 ms of CPU time and 0.4 ms of GPU time per step on KV selection and transfer, at least 6× lower than Quest, ShadowKV, and InfiniteHiP. These results validate that reusing each retrieved window throughout a sentence effectively amortizes KV retrieval costs.

Decoding throughput across context lengths. We evaluate how decoding throughput scales with context length using Qwen3-8B on a single H20 GPU under the inference engine of Section 3.2, with batch size 1 and a fixed KV budget of $B = 1024$. Following LongBench (Bai et al., 2024), we construct length-controlled contexts from the HotpotQA (Yang et al., 2018) validation split, ranging from 16K to 256K tokens, and generate 384 tokens per document. Figure 4 shows that KVScout maintains stable high throughput of 132–135 tokens/s across all context lengths. Compared with full-cache decoding, whose attention cost grows linearly with context length, KVScout achieves speedups growing to 5.3× at 256K. KVScout also outperforms Quest, ShadowKV, and InfiniteHiP at all context lengths, achieving respective speedups of 1.74×, 2.06×, and 3.65× at 256K. By combining a fixed KV budget with window reuse across multiple decoding steps (13.9 steps on average, Appendix D.2), KVScout bounds context-attention cost and amortizes KV selection and transfer costs, enabling high and stable decoding throughput.

Batched decoding. We evaluate KVScout's scalability by measuring aggregate decoding throughput across batch sizes at context lengths of 16K, 32K, and 64K, with a fixed KV budget of $B = 1024$. As shown in Figure 5, KVScout's throughput increases substantially with batch size across all three context lengths. Full-cache decoding runs out of GPU memory at batch sizes of 24 and 16 for 32K and 64K contexts, respectively. Notably, at 64K context, KVScout supports 3× larger batches, delivering 8.8× higher throughput than the maximum attainable throughput of full-cache decoding. KVScout also scales better than its baselines with increasing batch size, reaching speedups of 4.06–4.13× over Quest, 1.51–1.85× over ShadowKV, and 6.07–10.41× over InfiniteHiP at the largest evaluated batch size of 24 across the three context lengths. These results again demonstrate the efficiency of KVScout's bounded and reusable GPU-resident KV window.



(a) 16K context. (b) 32K context. (c) 64K context.

Figure 5: Decoding throughput with various batch sizes on Qwen3-8B at 16K, 32K, and 64K context lengths ($B = 1024$, single H20 GPU). Full-cache curves terminate at GPU memory limits.

Table 3: KV selection under sentence-level refresh ($B = 512$).

	LongBench Avg. ↑	
Method	**Qwen3-8B**	**DeepSeek-R1**
Quest	37.99	32.36
ShadowKV	42.76	33.35
KVScout	**48.35**	**39.30**

Table 4: Ablations on Qwen3-8B ($B = 512$). **Bold** marks the best result in each column.

Variant	**SQA**	**MQA**	**Sum.**	**Few.**	**Syn.**	**Code**	**Avg. ↑**
KVScout	**40.51**	**63.17**	**22.56**	**64.07**	**57.83**	**43.49**	**48.35**
w/o $\mathcal{S}$	15.29	13.68	19.56	18.99	22.18	16.09	17.45
w/o $\mathcal{D}_t$	38.55	49.60	20.64	61.39	53.34	36.87	43.19
w/o refinement	40.50	61.81	22.47	63.00	52.60	42.64	47.12

5.4 Ablation Studies

KV selection for multi-step reuse. We evaluate how effectively KVScout selects KV entries to support multiple subsequent decoding steps by comparing it with Quest and ShadowKV under sentence-level refresh at $B = 512$. As shown in Table 3, KVScout outperforms Quest and ShadowKV in average score on both evaluated models. KVScout also outperforms variants that replace only its dynamic-set selection with that of Quest or ShadowKV (Table 10). Beyond sentence-level refresh, Figure 2 compares KVScout with Quest, ShadowKV, and InfiniteHiP under fixed refresh intervals from 1 to 16 steps, where KVScout lies on the Pareto frontier of throughput and quality. In addition, the KVScout retriever achieves over 95% Recall@20 on held-out data (Appendix D.1). These results indicate that KVScout's predictive selection identifies context useful for subsequent generation, allowing the selected KV window to serve multiple decoding steps.

Effect of the static and dynamic sets. To study the roles of the static and dynamic sets, we construct KVScout (w/o $\mathcal{S}$) and KVScout (w/o $\mathcal{D}_t$), which remove the static and dynamic sets, respectively. In each variant, we reallocate the removed component's budget to the remaining set, keeping the total KV budget fixed at $B = 512$. As shown in Table 4, KVScout outperforms the "w/o $\mathcal{S}$" variant, showing that the static set retains context that remains essential at every decoding step. KVScout also outperforms the "w/o $\mathcal{D}_t$" variant, which falls below all KV retrieval baselines in Table 1, showing that retrieving additional context during generation further improves task quality. These results validate combining a static set with a dynamic set in KVScout's KV window design.

Effect of token refinement. To study the effect of token refinement, we construct a variant of KVScout (w/o refinement) that fills the dynamic set $\mathcal{D}_t$ with tokens from retrieved sentences in descending retrieval-score order until its budget B_d is filled. As shown in Table 4, KVScout outperforms its variant without candidate refinement, suggesting that distinguishing token importance within the retrieved sentences helps allocate the limited dynamic budget more effectively.

6 Conclusion

We propose KVScout, a sentence-level predictive KV retrieval method that reduces the KV retrieval frequency of long-context decoding while maintaining generation quality. At each generated-sentence boundary, KVScout learns to select the KV entries needed by the upcoming sentence and reuses them throughout its generation, amortizing KV selection and transfer costs over multiple decoding steps. Experiments show that KVScout achieves the highest average LongBench score among compressed methods across two models and three KV budgets, with $5.3\times$ the throughput of full-cache decoding at 256K context and up to $8.8\times$ under batched decoding at 64K context.

AI USE STATEMENT

In this work, we used generative AI models to construct the training data of our retriever: as described in Section 4.2, Qwen3-8B (Yang et al., 2025a) generates reasoning traces, and `gpt-5.6-luna` (OpenAI, 2026a) annotates the context sentences relevant to each reasoning sentence. We also used generative AI coding assistants to aid in implementing our method under the direction of the authors. We have not used generative AI tools for the other tasks with required disclosure, including developing theoretical models, formulating or proving mathematical claims, proposing hypotheses, designing research methodology or experiments, and interpreting results. Additionally, we used generative AI tools to search for relevant literature and to polish the writing of this paper. We have reviewed all AI-assisted work: we verified every AI-suggested reference against the original publication and checked all AI-polished text for technical accuracy. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work focuses on improving the decoding efficiency of long-context models and raises no ethical concerns that require disclosure. Our work involves no human subjects or personally identifiable information, and all experiments use publicly available models, datasets, and benchmarks.

REPRODUCIBILITY STATEMENT

We provide detailed implementation settings of KVScout in Section 5.1 and Appendix C. We will release both our code and training data upon acceptance.

REFERENCES

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

Paul C Bogdan, Uzay Macar, Neel Nanda, and Arthur Conmy. Thought anchors: Which llm reasoning steps matter? *arXiv preprint arXiv:2506.19143*, 2025.

Zefan Cai, Wen Xiao, Hanshi Sun, Cheng Luo, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, Animashree Anandkumar, Abedelkadir Asi, and Junjie Hu. R-KV: Redundancy-aware KV cache compression for reasoning models. In *Advances in Neural Information Processing Systems*, 2025a.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. In *Conference on Language Modeling*, 2025b.

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. In *Advances in Neural Information Processing Systems*, 2024.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, 2024.

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. In *Advances in Neural Information Processing Systems*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 2025.

Jitai Hao, Yuke Zhu, Tian Wang, Jun Yu, Xin Xin, Bo Zheng, Zhaochun Ren, and Sheng Guo. OmniKV: Dynamic context selection for efficient long-context LLMs. In *International Conference on Learning Representations*, 2025.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *Conference on Language Modeling*, 2024.

Luyang Huang, Shuyang Cao, Nikolaus Parulian, Heng Ji, and Lu Wang. Efficient attentions for long document summarization. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021.

Greg Kamradt. Needle in a haystack – pressure testing LLMs. GitHub repository, 2023. URL https://github.com/gkamradt/needle-in-a-haystack.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Heejun Lee, Geon Park, Jaduk Suh, and Sung Ju Hwang. InfiniteHiP: Extending language model context up to 3 million tokens on a single gpu. *arXiv preprint arXiv:2502.08910*, 2025.

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. In *Advances in Neural Information Processing Systems*, 2024.

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. In *International Conference on Learning Representations*, 2026.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

OpenAI. GPT-5.6 Luna. https://developers.openai.com/api/docs/models/gpt-5.6-luna, 2026a.

OpenAI. GPT-6 Astra. OpenAI API model documentation, 2026b. URL https://developers.openai.com/api/docs/models/gpt-6-astra. Accessed: 2026-09-20.

Matanel Oren, Michael Hassid, Nir Yarden, Yossi Adi, and Roy Schwartz. Transformers are multi-state RNNs. In *Conference on Empirical Methods in Natural Language Processing*, 2024.

Akshat Ramachandran, Marina Neseem, Charbel Sakr, Rangharajan Venkatesan, Brucek Khailany, and Tushar Krishna. Thinkv: Thought-adaptive kv cache compression for efficient reasoning models. In *International Conference on Learning Representations*, 2026.

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. In *International Conference on Machine Learning*, 2025.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context LLM inference. In *International Conference on Machine Learning*, 2024.

Wei Wu, Zhuoshi Pan, Kun Fu, Chao Wang, Liyi Chen, Yunchu Bai, Tianfu Wang, Zheng Wang, and Hui Xiong. TokenSelect: Efficient long-context inference and length extrapolation for LLMs via dynamic token-level KV cache selection. In *Conference on Empirical Methods in Natural Language Processing*, 2025.

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. In *Advances in Neural Information Processing Systems*, 2024a.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, 2024b.

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*, 2025a.

Lijie Yang, Zhihao Zhang, Zhuofu Chen, Zikun Li, and Zhihao Jia. Tidaldecode: Fast and accurate LLM decoding with position persistent sparse attention. In *International Conference on Learning Representations*, 2025b.

Qingyue Yang, Jie Wang, Xing Li, Zhihai Wang, Chen Chen, Lei Chen, Xianzhi Yu, Wulong Liu, Jianye Hao, Mingxuan Yuan, and Bin Li. AttentionPredictor: Temporal patterns matter for KV cache compression. In *Advances in Neural Information Processing Systems*, 2025c.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing*, 2018.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Conference on Empirical Methods in Natural Language Processing*, 2023a.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2O: Heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems*, 2023b.

Yuxuan Zhu, Ali Falahati, David H. Yang, and Mohammad Mohammadi Amiri. SentenceKV: Efficient LLM inference via sentence-level semantic KV caching. In *Conference on Language Modeling*, 2025.

A ALGORITHM

We provide the complete inference procedures of KVScout in Algorithm 1.

Algorithm 1 KVScout: Sentence-level predictive KV retrieval

Require: Input $\boldsymbol{x}$; trained retriever $\{\texttt{[SUM]}, \texttt{[RET]}, f_e, f_q\}$; layer ℓ; static budget B_s; dynamic budget B_d; window budget $B = B_s + B_d$; observation lengths m, w; temperature τ; threshold p.

Output: Generated sequence $\boldsymbol{y} = (y_0, y_1, \ldots)$.

// Prefilling: contextualized sentence indexing

Segment $\boldsymbol{x}$ into sentences $s_1, \ldots, s_N$;
$\mathcal{C} \leftarrow \textsc{Prefill}(\boldsymbol{x})$;
for $i = 1, \ldots, N$ **do**
  $\boldsymbol{h}^{(\ell)}_{\texttt{[SUM]}_i} \leftarrow \textsc{Probe}\big(\texttt{[SUM]}_i, \ell;\, s_1, \ldots, s_i\big)$;
  $\boldsymbol{e}_i \leftarrow f_e\big(\boldsymbol{h}^{(\ell)}_{\texttt{[SUM]}_i}\big)$;
end for
$\mathcal{E} \leftarrow \{\boldsymbol{e}_i\}_{i=1}^{N}$;
$\mathcal{S}, \mathcal{W}_0 \leftarrow$ top-B_s, top-B entries of $\mathcal{C}$ by attention from the last m tokens of $\boldsymbol{x}$;
Offload $\mathcal{C}$ to CPU;

// Decoding: predictive sentence retrieval

for $t = 0, 1, \ldots$ **until** $y_t = \texttt{<eos>}$ **do**
  $y_t \leftarrow \textsc{GenerateToken}\big(\boldsymbol{y}_{<t};\, \mathcal{W}_t\big)$;
  if y_t completes a sentence **then**
    // Sentence retrieval
    $\boldsymbol{h}^{(\ell)}_{\texttt{[RET]}_t} \leftarrow \textsc{Probe}\big(\texttt{[RET]}_t, \ell;\, \mathcal{W}_t, \boldsymbol{y}_{\le t}\big)$;
    $\boldsymbol{r}_t \leftarrow f_q\big(\boldsymbol{h}^{(\ell)}_{\texttt{[RET]}_t}\big)$;
    $(\pi_{t,1}, \ldots, \pi_{t,N}) \leftarrow \mathrm{softmax}\big(\mathrm{sim}(\boldsymbol{r}_t, \boldsymbol{e}_1)/\tau, \ldots, \mathrm{sim}(\boldsymbol{r}_t, \boldsymbol{e}_N)/\tau\big)$;
    $n_t^\star \leftarrow \min\big\{n : \sum_{k=1}^{n} \pi_{t,i_{(k)}} \ge p\big\}$;
    $\mathcal{A}_t \leftarrow \{s_{i_{(1)}}, \ldots, s_{i_{(n_t^\star)}}\}$;
    while $\sum_{s \in \mathcal{A}_t} |s| < B_d$ **do**
      $\mathcal{A}_t \leftarrow \mathcal{A}_t \cup \{s_{i_{(|\mathcal{A}_t|+1)}}\}$;
    end while
    // Token refinement
    Fetch the KV entries of sentences in $\mathcal{A}_t$ from CPU to GPU;
    $\mathcal{D}_{t+1} \leftarrow$ top-B_d fetched entries by attention from the last w decoding steps;
    $\mathcal{W}_{t+1} \leftarrow \mathcal{S} \cup \mathcal{D}_{t+1}$;
  else
    $\mathcal{W}_{t+1} \leftarrow \mathcal{W}_t$;
  end if
end for
return $\boldsymbol{y}$.

B CASE STUDIES

We examine whether the context that KVScout retrieves at a sentence boundary supports the sentence generated next. Figures 6 and 7 trace two instances in LongBench (Bai et al., 2024), a repository-level code completion task from RepoBench-P and a multi-hop question from 2WikiMQA. In each trace, round k denotes the k-th window refresh at a sentence boundary. Each round lists the preceding generated sentence, the top-3 retrieved context sentences with their retrieval weights, and the sentence generated with the refreshed window (marked by $\Rightarrow$).

Question: Which film has the director born later, The Wind'S Fierce or Special Delivery (1927 Film)?
Gold answer: The Wind'S Fierce

Round	Retrieval (weight)			Next sentence
7th	*... Then, in Passage 7, there's a mention of Mario Camus García, which I think is the same person. So Mario Camus is the director of The Wind's Fierce.*			⇒ Next, for Special Delivery (1927 Film), Passage 6 states that it's directed by Roscoe Arbuckle.
	#1	48.6%	Special Delivery is a 1927 American silent comedy film directed by Roscoe Arbuckle starring Eddie Cantor, Jobyna Ralston and William Powell.	
	#2	20.5%	Special Delivery (1927 film)	
	#3	14.7%	Special Delivery (French: C'est le bouquet !	
9th	*... Next, for Special Delivery (1927 Film), Passage 6 states that it's directed by Roscoe Arbuckle. Now, I need to check when Roscoe Arbuckle was born.*			⇒ Looking at Passage 5, which is about Roscoe Arbuckle, it says he was born on March 24, 1887.
	#1	68.8%	Roscoe Conkling "Fatty" Arbuckle (;	
	#2	29.0%	Passage 5: Roscoe Arbuckle	
	#3	1.4%	March 24, 1887 – June 29, 1933) was an American silent film actor, comedian, director, and screenwriter.	
11th	*... Looking at Passage 5, which is about Roscoe Arbuckle, it says he was born on March 24, 1887. Now, Mario Camus's birth date.*			⇒ Passage 7 mentions Mario Camus García was born on April 20, 1935.
	#1	71.0%	Passage 7: Mario Camus	
	#2	22.2%	Mario Camus García (20 April 1935 – 18 September 2021) was a Spanish film director and screenwriter.	
	#3	1.9%	Walter Augustus Barratt was born 3 June 1873 in Kilmarnock, the son of composer John Barratt;	

Figure 6: Retrieval trace of KVScout on a 2WikiMQA instance.

Observations. In both traces, the context retrieved at each round contains the fact that the next sentence states. The `[RET]` query is issued before this sentence is generated, so KVScout anticipates the evidence the upcoming sentence needs. In round 7 of the 2WikiMQA trace, for example, the preceding sentence concludes the first film, and KVScout already retrieves the director of the second film, which the next sentence then reports. As the reasoning advances, the needed evidence moves across the context, from the current file to a single repository class in the RepoBench-P trace and from passage to passage in the 2WikiMQA trace. KVScout relocates the window at every sentence boundary and keeps the supporting evidence of each sentence in the window. With this evidence, the model collects facts the answer requires and produces the gold answer in both cases.

C Additional Implementation Details

C.1 Retriever Architecture and Training

We keep the base model frozen and optimize only the `[SUM]` and `[RET]` input embeddings and the projection heads f_e and f_q. Both projection heads are two-layer MLPs with a hidden dimension of 1024 and a GELU (Hendrycks & Gimpel, 2016) activation, and their outputs are normalized to unit Euclidean norm. We set the retrieval layer to $\ell = 24$ for Qwen3-8B and $\ell = 21$ for DeepSeek-R1-Distill-Llama-8B, with retrieval dimension $d_r = 256$. For each document, we retain all positive sentences and randomly sample up to 128 negative sentences from the same context. We use AdamW (Loshchilov & Hutter, 2019) with constant learning rates of 10^{-3} for f_e and f_q and 2×10^{-4} for the probe embeddings, a weight decay of 0.01, a temperature τ of 0.05, and an effective batch size of 8 documents. We split the training data (Section 4.2) into a training set (95%) and a validation set (5%). Both the Qwen3-8B and DeepSeek-R1-Distill-Llama-8B retrievers are

Task: complete the next line of `Engine.init()` (RepoBench-P, Java); the context holds the repository's renderer classes and the current file.
Gold next line: `registerRenderer(Iterable.class, new DefaultIterableRenderer());`

Round	Retrieval (weight)			Next sentence
3rd	*... The user is asking for the next line of code in the Engine class's init method.*			⇒ The current code has registered renderers for Object, Map, and Collection.
	#1	41.0%	`registerRenderer(Collection.class`	
	#2	32.9%	`registerRenderer(Map.class`	
	#3	5.2%	`registerRenderer(Object.class`	
5th	*... The next logical step would be to register the remaining renderers mentioned in the existing code.*			⇒ Looking at the other renderer classes in the imports, there's DefaultIterableRenderer and DefaultMapRenderer.
	#1	17.9%	`} } src/com/floreysoft/jmte/renderer/DefaultMapRenderer.java`	
	#2	17.3%	`} src/com/floreysoft/jmte/renderer/DefaultIterableRenderer.java`	
	#3	7.8%	`@SuppressWarnings("unchecked")`	
7th	*... Wait, the existing code already registered Map and Collection.*			⇒ Wait, the user provided code for DefaultIterableRenderer, which is a Renderer<Iterable>.
	#1	23.8%	`} src/com/floreysoft/jmte/renderer/DefaultIterableRenderer.java`	
	#2	17.4%	`public class DefaultIterableRenderer implements Renderer<Iterable> {`	
	#3	13.1%	`@SuppressWarnings("unchecked")`	

Figure 7: Retrieval trace of KVScout on a RepoBench-P instance.

trained on the training set for 18 epochs, which takes about 100 GPU hours on NVIDIA H20 GPUs per retriever.

C.2 Sentence-Boundary Detection

KVScout uses one rule-based sentence segmenter in three places: segmenting the input into the sentences $s_1, \ldots, s_N$ indexed during prefill (Section 4.1), segmenting reasoning traces into target sentences when constructing training annotations (Section 4.2), and detecting generated-sentence boundaries that trigger retrieval during decoding. Because annotation and decoding use the same rule, the retriever is queried at inference at the same kind of boundary at which it is supervised.

Segmentation rule. Let $\Sigma = \{\texttt{.}, \texttt{!}, \texttt{?}, \texttt{;}, \texttt{\textbackslash n}\}$ be the delimiter set, where `\n` denotes a line break, and let $L_{\min} = 4$ and $L_{\max} = 24$ be the minimum and maximum sentence lengths in tokens. We call a token a delimiter token if it contains a character in Σ. The segmenter scans the token sequence from left to right and ends the current sentence at the first delimiter token such that the span from the start of the sentence through this token contains at least $L_{\min}$ tokens. Delimiter tokens that occur before the span reaches $L_{\min}$ tokens do not end a sentence. If the span reaches $L_{\max}$ tokens before such a delimiter token occurs, the segmenter ends the sentence after the $L_{\max}$-th token. The next sentence begins at the token immediately after the boundary, and any tokens after the last boundary form the final sentence. The minimum length prevents short fragments, such as enumeration markers (`1.`), blank lines, and short lines of code, from forming stand-alone sentences. The maximum length splits long spans without delimiters, such as code blocks, tables, and run-on text, into multiple sentences of bounded size. The same Σ, $L_{\min}$, and $L_{\max}$ are used for all models, benchmarks, and tasks.

C.3 Annotation Prompt

We construct the sentence-level annotations in Section 4.2 by querying `gpt-5.6-luna` with the prompt in Figure 8. The placeholder {`context`} is replaced with the input context, and {`trace`} with the reasoning trace split into sentences, one per line and numbered as `1)`, `2)`, ... Each

Table 5: Retrieval recall (%) of the KVScout retriever on the validation set.

Method	Recall@1	Recall@5	Recall@10	Recall@20
Random	0.4	1.8	3.5	7.0
KVScout (Qwen3-8B)	42.0	81.2	91.0	96.2
KVScout (DeepSeek-R1-Distill-Llama-8B)	38.9	78.4	89.1	95.4

returned quote is located in the input context by string matching, and the context sentences it covers are labeled as relevant to the corresponding target sentence.

```
You are given a CONTEXT (reference text) and a reasoning TRACE split
into numbered sentences. For each trace sentence that is about a
specific fact/entity the CONTEXT has information on, quote the
COMPLETE context sentence(s) that state the evidence for it.

How to quote:
- ALWAYS quote whole sentences, copied VERBATIM from the CONTEXT —
  from the first word of the sentence to its final period. Never a
  fragment, a bare date, or a bare name.
- Prefer ONE sentence per piece of evidence; use two CONSECUTIVE
  sentences only when the fact is genuinely split across them (e.g., a
  pronoun resolved by the previous sentence).
- Pick the sentence that ESTABLISHES the fact — the one naming the
  entities involved — not merely a sentence where the answer string
  happens to appear. For summary/conclusion steps ("so the answer is
  X"), quote the same source sentence(s) that support X, exactly as
  for the step that first derived X.
- A step may draw on several distinct facts (multi-hop) — one quote
  per fact.

IMPORTANT — annotate the evidence, do NOT verify:
- Return the evidence sentence EVEN WHEN the trace sentence states the
  fact WRONG. The goal is to locate what the step is drawing on, not
  to check it.
- OMIT a trace sentence ONLY if the CONTEXT contains nothing about
  what it is talking about (pure planning / meta-reasoning, or facts
  the context never mentions).

Return JSON only: {"steps": [{"sent": <int>,
  "quotes": ["<verbatim full sentence(s) from CONTEXT>", ...]}]}

CONTEXT:
{context}

TRACE (numbered sentences):
{trace}
```

Figure 8: Prompt used to annotate the context sentences relevant to each target sentence.

D Additional Experimental Results

D.1 Retrieval Accuracy

To evaluate whether the retriever identifies the context needed by the next sentence before it is generated, we test the retriever trained for each backbone on the held-out validation split of the training data (Section 4.2). We rank all context sentences by $\pi_{t,i}$ in Equation (4), computed at the boundary preceding the target sentence, and report Recall@k, the fraction of relevant sentences ranked within the top k, averaged over target sentences.

Table 6: Average refresh interval on LongBench with Qwen3-8B. The intervals of Quest and InfiniteHiP are fixed by design.

B	Average refresh interval (tokens)		
	Quest	InfiniteHiP	KVScout
512	1	4	**13.6**
1024	1	4	**13.9**
2048	1	4	**13.9**

As shown in Table 5, KVScout achieves high recall on both backbones and far outperforms a random ranking at every k. At Recall@20, KVScout recovers over 95% of the relevant sentences, which makes its retriever reliable enough to accelerate decoding. These results show the effectiveness of our retriever design and query-sentence contrastive objective.

D.2 Refresh Frequency

To quantify how often KVScout updates its KV cache during decoding, we measure the average refresh interval, defined as the average number of generated tokens between consecutive refreshes, for Qwen3-8B on LongBench, and compare it with Quest (Tang et al., 2024) and InfiniteHiP (Lee et al., 2025) in Table 6. As shown in Table 6, KVScout refreshes once every 13.6 to 13.9 generated tokens across all budgets. This interval is about $14\times$ that of Quest, which refreshes at every decoding step, and more than $3\times$ that of InfiniteHiP, which refreshes every 4 steps. Each refresh reselects the KV entries used for attention, so a longer interval directly reduces the selection and transfer overhead per generated token.

D.3 Sensitivity Analysis

Effect of B_d. To study the budget allocation between the static and dynamic sets, we vary the dynamic-set budget B_d on HotpotQA using Qwen3-8B, with the total KV budget fixed at $B = 1024$ and the static-set budget $B_s = B - B_d$. As shown in Figure 9, KVScout maintains stable performance for B_d from 128 to 896, showing its robustness to a wide range of allocations. Performance degrades only at the two extremes, dropping slightly at $B_d = 0$, where no dynamic set is retrieved, and sharply at $B_d = 1024$, where the static set is entirely removed. We therefore use $B_d = 640$ for $B = 1024$ by default, well within the stable range.

Effect of w. We study the effect of different observation-window sizes for candidate refinement using Qwen3-8B on HotpotQA. As shown in Figure 10, KVScout is robust to different window sizes, with F1 scores fluctuating slightly around 68.5 across 1 to 32. In practice, we recommend using a small observation window ($w \leq 32$), which is sufficient to identify important tokens during candidate refinement.

Effect of p. Figure 11 shows the effect of varying the retrieval threshold p on KVScout's performance on HotpotQA using Qwen3-8B, together with the candidate fraction $n_t^\star/N$ averaged over all retrieval steps. As can be seen, performance generally improves as p increases to 0.95 and remains stable at higher thresholds, showing that retrieving a broader set of potentially relevant sentences helps preserve useful evidence for subsequent candidate refinement. Meanwhile, the fraction of retrieved sentences grows with p and stays within about 5% for $p \leq 0.99$. We therefore recommend setting $p \in [0.95, 0.99]$ and use $p = 0.99$ by default.

D.4 Per-Task LongBench Results

Tables 7 and 8 report the full per-task results of Table 1 on Qwen3-8B and DeepSeek-R1-Distill-Llama-8B, respectively. As shown, KVScout achieves the highest average score among compressed methods under all three KV budgets on both models.

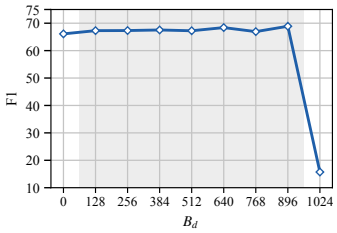


Figure 9: Effect of B_d

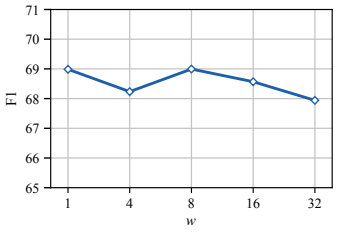


Figure 10: Effect of w

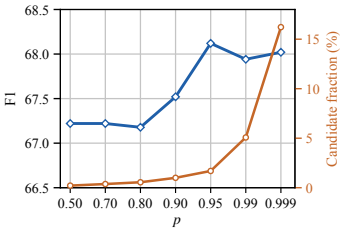


Figure 11: Effect of p

Table 7: Per-task LongBench results on Qwen3-8B, grouped by KV budget B. Group averages are reported in Table 1. **Bold** indicates the best compressed result within each budget.

		Single-Doc QA			Multi-Doc QA			Summarization			Few-shot			Synthetic		Code		
	Method	NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	Musique	GovRep	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PRe	Lcc	RB-P	Avg.
	Full cache	26.8	46.4	48.6	69.2	76.9	47.3	28.3	19.5	22.7	72.5	87.3	32.8	28.7	100.0	47.5	46.2	50.0
512	SnapKV	26.0	41.7	48.0	60.4	53.6	34.8	22.6	19.1	20.2	71.5	80.3	32.3	12.2	94.5	37.4	36.3	43.2
	Ada-KV	26.0	41.3	47.4	63.2	53.1	31.1	22.5	19.0	20.0	72.5	82.5	31.9	8.1	95.0	37.8	34.8	42.9
	Quest	24.6	43.2	**50.1**	66.8	74.6	47.0	27.2	18.8	23.0	69.0	86.2	32.7	**20.2**	**99.0**	**48.6**	41.9	**48.3**
	AttentionPredictor	**28.7**	45.5	48.8	65.7	74.3	44.4	26.0	19.5	22.4	70.0	85.7	31.8	11.8	95.5	43.5	42.3	47.2
	ShadowKV	27.7	**45.8**	49.9	63.0	**77.0**	44.4	**28.4**	**19.9**	**23.6**	71.5	**87.2**	**33.1**	14.0	91.0	43.9	43.2	47.7
	SentenceKV	26.3	42.5	47.4	61.7	66.9	39.3	25.8	19.1	22.2	73.5	83.7	32.8	13.6	98.5	43.2	39.3	46.0
	InfiniteHiP	27.3	41.8	48.5	65.9	71.4	41.4	25.3	19.0	22.6	65.5	86.6	31.8	12.3	96.0	45.4	**44.6**	46.6
	KVScout	26.7	45.5	49.3	**67.5**	73.7	**48.3**	25.7	19.8	22.2	**74.5**	85.4	32.3	18.7	97.0	44.8	42.2	**48.3**
1024	SnapKV	28.4	44.2	47.4	66.1	65.9	42.1	24.9	19.6	21.5	**76.5**	83.0	31.8	16.6	99.0	43.7	39.5	46.9
	Ada-KV	28.9	44.3	48.5	64.5	64.8	41.0	24.9	19.8	21.8	75.0	83.1	**33.6**	14.0	99.0	42.3	37.6	46.4
	Quest	26.8	44.9	**50.6**	66.5	75.6	45.0	27.6	19.5	22.8	73.5	87.2	32.9	**25.6**	99.0	**48.3**	**46.4**	49.5
	AttentionPredictor	25.6	43.9	**50.6**	**69.4**	75.8	**49.0**	27.2	19.7	22.5	73.0	88.0	32.6	18.4	99.5	46.8	45.5	49.2
	ShadowKV	28.9	43.7	48.1	65.2	76.2	44.9	**28.6**	**20.2**	**23.1**	75.0	86.1	33.5	14.5	99.0	45.7	43.4	48.5
	SentenceKV	**29.8**	**46.3**	49.0	63.3	74.6	45.0	27.2	19.7	23.0	74.5	84.4	32.4	20.9	**100.0**	45.5	42.7	48.6
	InfiniteHiP	26.9	45.4	49.0	66.9	76.5	44.6	27.0	19.3	22.8	72.5	**89.0**	31.5	19.6	99.0	46.2	45.3	48.8
	KVScout	27.5	45.8	49.4	68.4	**77.1**	48.9	27.0	19.7	22.8	75.5	87.1	32.3	23.1	99.0	44.9	45.1	**49.6**
2048	SnapKV	28.6	44.6	49.8	66.7	75.5	44.0	26.8	19.7	22.6	73.0	84.3	32.9	22.5	99.0	45.7	42.7	48.7
	Ada-KV	**29.2**	45.6	49.5	67.1	72.4	45.6	26.8	19.3	22.6	74.5	84.3	32.9	27.5	99.5	46.3	42.1	49.1
	Quest	27.5	**47.7**	49.9	67.0	**76.7**	46.3	27.9	19.7	22.7	73.0	86.0	32.9	27.5	**100.0**	47.4	**46.4**	49.9
	AttentionPredictor	27.3	46.7	50.2	68.8	**76.7**	**49.3**	27.7	19.2	22.7	**75.0**	86.8	33.4	24.8	**100.0**	46.8	46.3	50.1
	ShadowKV	28.9	43.5	**50.3**	65.1	75.3	45.2	**28.5**	**20.0**	**23.2**	73.0	87.9	**34.6**	13.5	**100.0**	45.5	41.9	48.5
	SentenceKV	28.0	45.5	49.7	67.7	75.2	47.3	27.9	19.3	22.9	74.0	87.3	33.6	26.1	**100.0**	46.4	43.3	49.6
	InfiniteHiP	28.3	45.0	49.7	66.6	76.0	46.7	27.4	19.3	22.9	73.0	88.1	32.9	25.1	**100.0**	46.0	**46.4**	49.6
	KVScout	26.7	46.6	49.9	**70.4**	76.1	48.1	27.6	19.7	22.7	74.0	**88.3**	33.8	**28.0**	**100.0**	**48.4**	45.8	**50.4**

D.5 Additional Needle-in-a-Haystack Results

Figure 12 extends the Needle-in-a-Haystack evaluation in Section 5.2 to Qwen3-8B and DeepSeek-R1-Distill-Llama-8B at budgets B=1024 and B=2048. As shown, KVScout stays close to Full at both budgets and outperforms Quest in every setting.

D.6 Full result of Table 3

Table 9 shows the full result of Table 3.

D.7 Replacing the Dynamic-Set Selection

To isolate the effect of how KVScout selects the dynamic set $\mathcal{D}_t$, we keep its KV window design and sentence-level refresh and replace only its retriever with the selection rule of Quest or ShadowKV. Table 10 shows the results on the HotpotQA and PassageCount tasks of LongBench (Bai et al., 2024), using Qwen3-8B with $B = 512$ ($B_s = 320$, $B_d = 192$) and the same static set $\mathcal{S}$ for all variants. As can be seen, KVScout outperforms the Quest and ShadowKV variants by 7.61 and 4.25 points on HotpotQA and by 5.47 and 2.57 points on PassageCount. This shows that KVScout selects the context required by upcoming sentences more accurately, so a single selection supports decoding across multiple sentences.

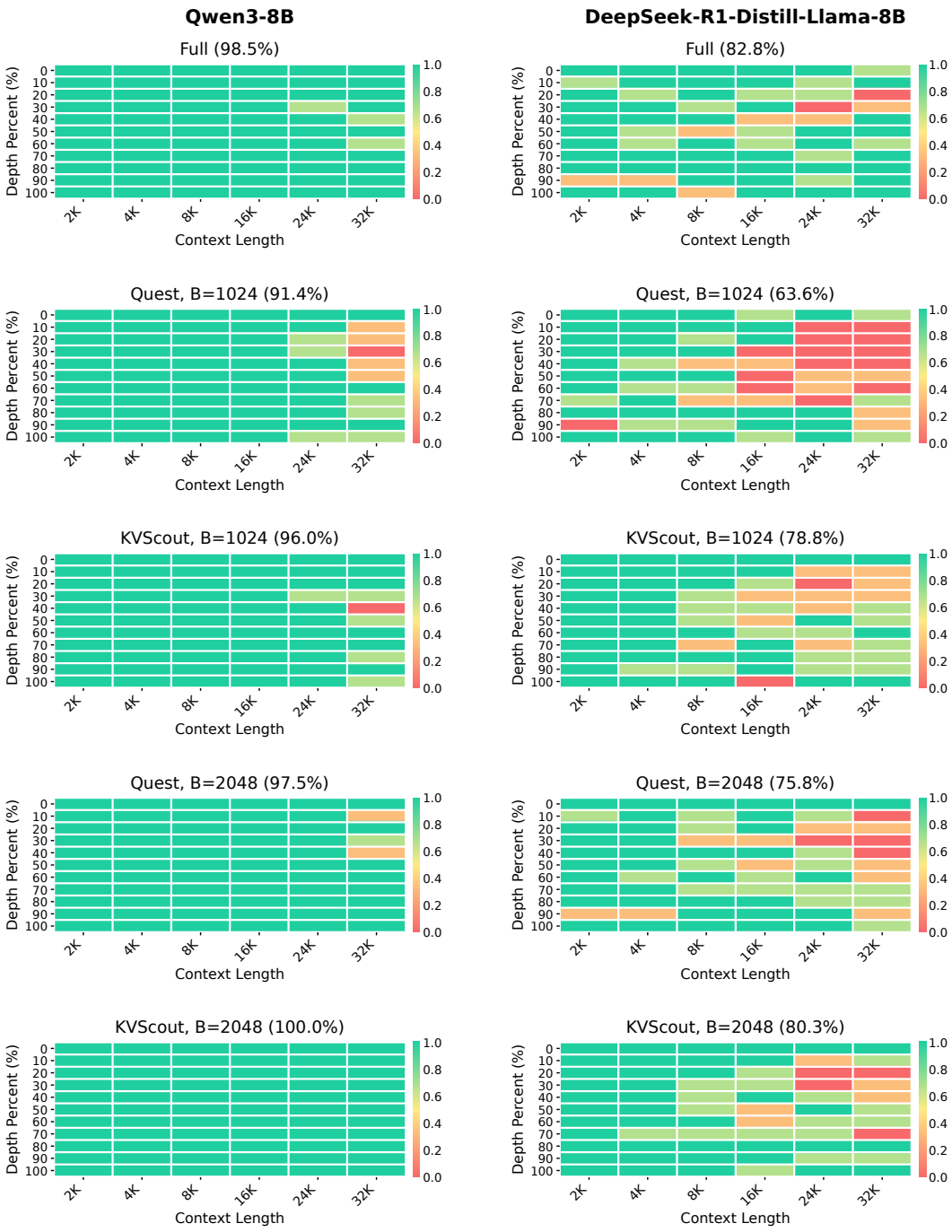


Figure 12: Needle-in-a-Haystack results on Qwen3-8B (left) and DeepSeek-R1-Distill-Llama-8B (right). The first row shows full-cache decoding, and the remaining rows show Quest and KVScout with $B = 1024$ and $B = 2048$.

Table 8: Per-task LongBench results on DeepSeek-R1-Distill-Llama-8B, grouped by KV budget B. Group averages are reported in Table 1. **Bold** indicates the best compressed result within each budget.

		Single-Doc QA			Multi-Doc QA			Summarization			Few-shot			Synthetic		Code		
	Method	NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	Musique	GovRep	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PRe	Lcc	RB-P	Avg.
	Full cache	20.0	31.4	44.3	43.5	55.8	42.2	27.1	20.4	24.3	60.0	85.4	33.5	14.5	98.0	30.8	29.7	41.3
512	SnapKV	18.9	26.1	40.8	43.6	39.0	30.0	21.1	20.1	21.6	58.5	79.3	32.1	6.1	81.5	27.6	28.7	35.9
	Ada-KV	20.0	26.3	41.6	42.4	37.2	31.6	22.3	20.2	21.6	56.5	81.0	32.4	4.2	75.4	26.5	26.1	35.3
	Quest	20.2	**31.4**	41.5	45.1	53.7	37.0	25.8	19.9	23.8	52.5	**85.8**	32.0	9.7	79.7	29.1	30.3	38.6
	AttentionPredictor	19.8	29.2	43.0	**46.9**	**58.2**	**44.3**	26.4	20.3	23.8	54.5	82.1	31.9	4.9	70.3	27.0	28.1	38.2
	ShadowKV	20.5	30.6	42.7	46.4	53.2	41.5	**27.4**	19.5	**24.1**	**60.5**	82.8	31.5	**11.1**	78.3	27.9	27.4	39.1
	SentenceKV	**21.9**	27.8	43.2	43.1	48.5	35.1	24.6	20.0	23.7	59.0	82.4	**32.8**	8.9	73.6	27.8	30.1	37.7
	InfiniteHiP	19.3	27.2	**44.1**	46.2	54.0	40.1	24.9	20.1	23.5	53.5	84.4	31.3	5.7	63.7	30.0	**30.6**	37.4
	KVScout	20.3	29.6	40.9	45.5	55.6	41.3	24.5	**20.6**	23.2	57.5	82.2	31.5	10.6	**84.5**	**30.7**	30.0	**39.3**
1024	SnapKV	21.7	28.9	41.5	43.5	47.0	36.0	23.8	**20.8**	23.2	60.0	84.4	32.5	12.2	88.6	29.1	28.5	38.8
	Ada-KV	21.8	29.0	40.7	44.8	47.5	36.1	23.9	20.5	23.3	61.0	81.6	33.2	9.2	89.1	29.1	28.6	38.7
	Quest	20.8	**33.1**	44.5	42.8	56.2	39.2	26.3	19.9	**24.1**	56.5	84.7	**33.3**	14.5	**92.6**	29.7	30.4	40.5
	AttentionPredictor	**23.8**	30.6	43.6	47.4	56.2	**45.4**	26.6	20.2	24.0	56.5	83.5	33.0	11.3	87.3	29.1	31.0	40.6
	ShadowKV	21.3	29.0	40.7	46.4	55.9	43.7	**27.1**	19.8	24.0	60.0	84.5	32.0	11.6	88.0	27.2	28.0	39.9
	SentenceKV	20.8	29.8	43.1	45.0	**58.3**	38.4	25.9	20.6	24.0	61.0	83.1	32.8	13.2	87.5	29.8	29.9	40.2
	InfiniteHiP	19.5	30.7	44.2	44.3	55.4	41.8	26.2	19.8	24.0	**61.2**	83.8	32.6	9.7	87.2	**31.6**	**31.7**	40.2
	KVScout	21.7	31.4	**45.9**	**47.6**	56.8	41.5	26.2	20.6	24.0	58.5	**85.0**	32.9	**16.1**	87.3	31.0	29.4	**41.0**
2048	SnapKV	20.5	29.9	42.7	44.7	52.0	37.8	25.5	20.5	23.8	63.0	83.6	32.9	14.6	92.0	30.3	30.1	40.2
	Ada-KV	21.0	28.2	43.0	45.4	55.3	36.7	25.5	20.8	23.8	63.0	84.8	**33.6**	11.1	90.2	29.8	31.3	40.2
	Quest	21.4	31.2	43.5	46.2	57.4	39.9	26.3	20.3	24.0	60.0	**85.2**	**33.6**	11.6	**98.7**	29.7	30.5	41.2
	AttentionPredictor	22.2	30.4	44.1	46.3	56.9	39.5	26.1	20.2	24.1	65.0	84.6	33.4	13.1	95.2	30.1	**32.3**	41.5
	ShadowKV	22.1	29.2	42.7	42.1	54.1	39.9	**26.8**	**20.9**	24.1	62.5	84.6	31.8	9.5	95.2	28.6	28.0	40.1
	SentenceKV	21.8	29.9	42.1	44.4	56.1	**41.0**	**26.8**	20.4	**24.4**	**65.5**	84.2	32.7	11.6	94.2	**31.2**	30.8	41.1
	InfiniteHiP	**22.3**	29.3	**44.6**	47.0	54.7	40.8	**26.8**	20.3	24.1	59.0	84.5	32.9	13.0	91.6	30.3	29.1	40.6
	KVScout	21.7	**31.9**	43.7	**47.2**	**57.7**	40.7	26.6	20.7	24.2	63.7	82.2	32.9	**17.6**	96.0	30.2	30.9	**41.7**

Table 9: KV-selection quality on LongBench under sentence-level refresh ($B = 512$). **Bold** marks the best result within each model.

	SQA	MQA	Summary	Few-shot	Synthetic	Code	Average ↑
			Qwen3-8B				
Quest	34.62	41.48	20.97	54.57	43.61	32.80	37.99
ShadowKV	36.25	54.98	**22.59**	59.59	42.00	39.96	42.76
KVScout	**40.51**	**63.17**	22.56	**64.07**	**57.83**	**43.49**	**48.35**
			DeepSeek-R1-Distill-Llama-8B				
Quest	27.53	32.40	21.49	50.45	33.34	27.75	32.36
ShadowKV	27.04	37.51	21.85	53.81	30.81	25.69	33.35
KVScout	**30.26**	**47.46**	**22.80**	**57.10**	**47.55**	**30.39**	**39.30**

Table 10: Replacing the dynamic-set selection of KVScout on Qwen3-8B ($B = 512$). All variants share the same static set $\mathcal{S}$.

Selection of $\mathcal{D}_t$	**HotpotQA**	**PassageCount**
Quest	59.92	13.19
ShadowKV	63.28	16.09
KVScout	**67.53**	**18.66**