LAZYRECALL : DECISION-RISK-AWARE ONLINE KV CACHE MANAGEMENT FOR LONG-FORM GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-form generation and extended reasoning continuously expand the key–value (KV) cache, making GPU memory a bottleneck for scalable inference. Managing a fixed GPU KV budget requires three coupled decisions: *what to evict, when to evict, and what if an evicted KV becomes relevant again*. Existing methods often select KVs using attention statistics or local removal effects, which may not provide a globally comparable measure of their impact on the final model decision. Even with a precomputed eviction order, frequent rescoring incurs overhead, while bulk eviction removes states before their capacity is needed. Moreover, once evicted states are permanently discarded, subsequent rescoring cannot recover them if their relevance returns.

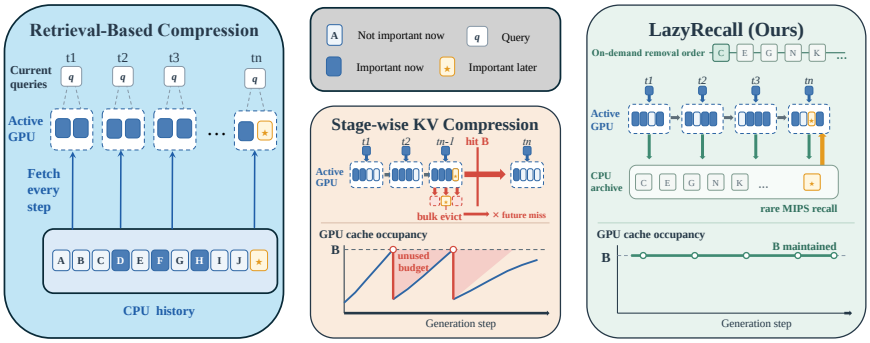
We introduce *LazyRecall*, an online KV-cache management framework that jointly addresses selection, execution, and recovery. *Decision-Risk* uses a shared vector–Jacobian product (VJP) to approximate each KV's effect on the final model decision, enabling model-wide selection on a common risk scale. *On-Demand Removal* decouples periodic planning from on-demand execution, retaining KVs until their capacity is required. *Asynchronous Recall* keeps evicted states recoverable on the CPU and retrieves relevant history for subsequent re-evaluation without blocking current decoding. Experiments on long-form writing and mathematical reasoning show that LazyRecall reduces the active GPU KV cache to as little as 15.5% of FullCache while maintaining comparable generation quality, with negligible additional per-request decoding latency. Anonymous code is available at `https://anonymous.4open.science/r/LazyRecall/`.

1 INTRODUCTION

Large language model (LLM) applications are expanding from short-form question answering to long-document understanding (Bai et al., 2024), repository-scale software engineering (Jimenez et al., 2024), multi-turn agent interaction (Liu et al., 2024b), and long-chain reasoning that may require thousands of reasoning tokens (Guo et al., 2025). As decoding proceeds, generated content becomes part of the context and may influence model decisions much later. Long-context inference thus requires continuous maintenance of model execution state, not merely processing longer inputs.

The KV cache is a major source of dynamic GPU memory usage during autoregressive generation. It grows with sequence length and is repeatedly accessed, constraining generation length, service concurrency, and inference efficiency (Kwon et al., 2023; Zhang et al., 2023; Lee et al., 2024). We study online KV management under a user-specified active GPU KV budget B: once the cache reaches this limit, admitting new KVs requires evicting an equivalent amount of existing KV state. Existing methods broadly follow two approaches. Eviction-based methods maintain a bounded GPU cache by selecting which states to retain (Zhang et al., 2023; Liu et al., 2023; Li et al., 2024; Cai et al., 2024; Fu et al., 2025; Cai et al., 2026). They avoid repeated access to offloaded history, but permanently discarded KVs are no longer available to subsequent decoding. Retrieval-based methods instead retain historical KVs on the CPU and access a query-dependent subset (Lee et al., 2024; Zhang et al., 2025; Liu et al., 2026; Wu et al., 2026), preserving access to past information at a recurring cost in search, computation, or transfer (Figure 1(a)).

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.



(a) Per-step CPU retrieval (b) Stage-wise bulk compression (c) On-demand removal with recall

Figure 1: KV-cache management under a fixed GPU budget B. (a) Retrieval-based attention repeatedly accesses CPU-resident history. (b) Stage-wise compression evicts a batch when the cache reaches its budget, leaving capacity unused during refill and losing access to discarded states. (c) LazyRecall periodically plans an eviction order, executes it incrementally, and asynchronously recalls states from the CPU, keeping the logical active cache at B after saturation.

We argue that online KV management under a fixed GPU budget involves three coupled questions: *what to evict, when to evict, and what if an evicted KV becomes relevant again?* First, **what to evict?** Existing eviction methods often select KVs using attention statistics or removal-induced changes in intermediate representations (Zhang et al., 2023; Liu et al., 2023; Li et al., 2024; Shen et al., 2025). Budget allocation across layers and heads can further constrain model-wide selection (Cai et al., 2024; Fu et al., 2025), while local removal criteria do not directly measure how a perturbation propagates to the final model decision. Second, **when to evict?** Incremental management can continuously utilize cache capacity but incurs repeated scoring and selection overhead, whereas stage-wise compression amortizes this cost through bulk eviction but releases capacity that could still retain historical states (Figure 1(b)). Third, **what if relevance returns?** Persistence in attention patterns can guide KV selection (Liu et al., 2023; Li et al., 2024), but does not guarantee that discarded states will remain irrelevant (Xu et al., 2025; Zhao et al., 2026). Once permanently deleted, such states cannot be recovered by subsequent rescoring.

To address these three questions, we introduce *LazyRecall* (Figure 1(c)), an online KV-management framework that integrates model-wide risk-based selection, on-demand eviction, and historical recovery. For *what to evict*, *Decision-Risk* estimates how removing each physical KV affects the final model decision. Each audit uses a standard vector–Jacobian product (VJP) (Baydin et al., 2018) to propagate downstream decision sensitivity to internal states in a single shared backward pass, providing a common approximate risk score for model-wide selection. For *when to evict*, *On-Demand Removal* decouples eviction planning from execution: a model-wide eviction order is constructed periodically but consumed only as new KVs require capacity, avoiding both per-step rescoring and premature capacity release. Finally, for *what if relevance returns*, *Asynchronous Recall* retains evicted KVs on the CPU and performs low-frequency maximum inner-product search (MIPS) asynchronously with GPU decoding. Retrieved historical states become candidates for subsequent risk audits rather than being required by the current attention computation, keeping retrieval off the current decoding path.

Extensive experiments on long-form writing and mathematical reasoning show that LazyRecall preserves generation quality close to FullCache while keeping as little as 15.5% of the active KVs on the GPU, with negligible additional per-request decoding latency.

Our main contributions are as follows:

- We identify three coupled decisions in online KV management: *what to evict, when to evict, and what if relevance returns*, motivating globally comparable selection, on-demand eviction, and historical recovery.
- We introduce *LazyRecall*, an online KV-management framework that integrates *Decision-Risk* for model-wide selection, *On-Demand Removal* for on-demand execution, and *Asynchronous Recall* for recovering relevant historical KVs without blocking current decoding.

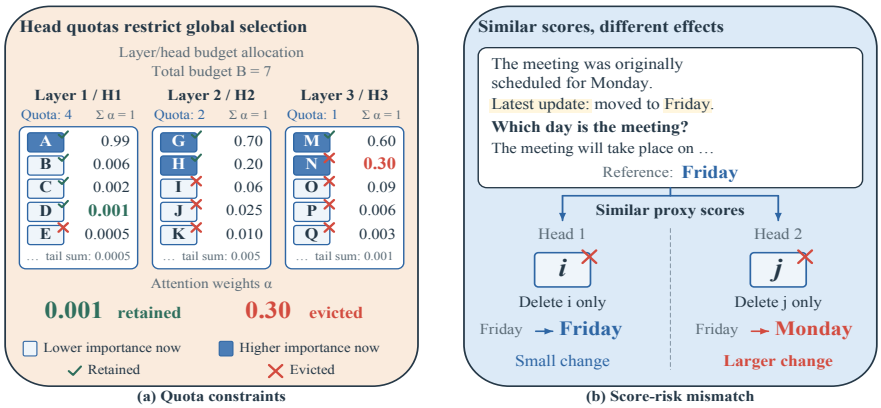


Figure 2: Why global KV selection requires a common objective? (a) Local quotas constrain cross-head selection. (b) Similar local proxy scores need not imply similar effects on the final prediction.

- We extensively evaluate LazyRecall across multiple model families on long-form writing and mathematical reasoning, demonstrating strong generation quality under substantially reduced GPU KV budgets with low decoding overhead.

2 Design Rationale

Online KV management under a fixed GPU budget involves three coupled decisions: **selection**—what should be evicted; **execution**—when should the selected states actually leave the active cache; and **recovery**—what if an evicted state becomes relevant again? We examine these questions separately to motivate the subsequent design.

2.1 Global KV Selection Requires A Common Objective

Existing KV management methods often decompose a model-wide cache budget into layer- or head-specific quotas and then select KVs locally. For example, SnapKV assigns equal retention capacities across heads, whereas PyramidKV and HeadKV adapt budgets across layers or heads (Li et al., 2024; Cai et al., 2024; Fu et al., 2025). Once these quotas are fixed, however, they constrain selection: a more important KV in one head may be evicted while a less important one remains in another, as illustrated in Figure 2(a).

Removing fixed quotas alone does not solve the problem. Global selection also requires scores across layers and heads to have a *common meaning*. LAVa (Shen et al., 2025) and OBCache (Gu et al., 2025), for example, characterize removal-induced changes in intermediate attention outputs. Although these scores provide a common basis for comparison, such local perturbations do not directly capture how removal propagates through the downstream network to affect the final prediction. Consequently, KVs with similar local scores may have substantially different prediction-level effects, as illustrated in Figure 2(b).

A natural common objective is the change in the model's output distribution caused by removing a KV. At a fixed decoding checkpoint, let p denote the original predictive distribution and $p^{(-i)}$ the distribution after independently removing candidate i. We define its instantaneous deletion risk as

$$\mathcal{K}_i = D_{\mathrm{KL}}\left(p \,\middle\|\, p^{(-i)}\right). \tag{1}$$

A smaller $\mathcal{K}_i$ indicates a safer deletion, and evaluating all candidates in the same output space makes the objective comparable across layers and physical KV heads. Exact evaluation, however, requires a separate downstream recomputation for every candidate and is impractical during decoding.

Figure 3 provides a diagnostic comparison against measured deletion risk, obtained by independently removing candidate KVs and recomputing the predictive distribution. Compared with the local removal-aware criteria LAVa and OBCache, our downstream-aware Decision-Risk (Section 3.1)

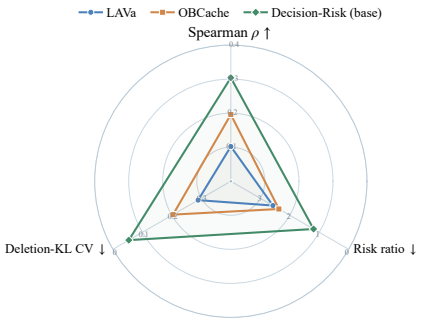


Figure 3: Diagnostic comparison of KV scoring criteria against measured deletion risk using CV, Spearman's ρ, and the deletion-risk ratio.

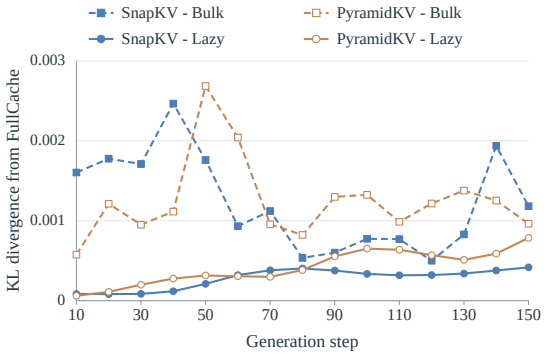


Figure 4: Predictive KL divergence from FullCache during cache refill. Dashed and solid lines denote Bulk and Lazy execution, respectively; lower is better.

shows better risk consistency and ranking agreement. Moreover, for joint removal of a fixed number of KVs from the same candidate pool, we use the minimum joint-deletion KL found by exhaustive search as a restricted oracle. The deletion-risk ratio is 1.194 for *Decision-Risk*, compared with 2.572 for LAVa and 2.370 for OBCache. These results motivate our first design requirement: *global KV selection needs an efficient, globally comparable approximation of each KV's effect on the final model decision*. Metric definitions and experimental details are provided in Appendix B.

2.2 Eviction Decisions Need Not Be Executed Immediately

Incremental eviction can fully utilize cache capacity, but rescoring and sorting at every decoding step introduce substantial management overhead. Stage-wise compression amortizes this cost through bulk eviction, but prematurely removes states whose capacity is not yet needed. These are not the only two choices: *eviction planning and execution can be decoupled*. An eviction order can be determined periodically and reused across steps, while its candidates are removed only as new KVs require capacity.

The budget B is measured in token-equivalent units. A physical KV slot stores a pair of key and value vectors for one position in one layer and physical KV head. For a model with L layers and H physical KV heads per layer, B corresponds to $B_{\text{phys}} = B \times L \times H$ physical KV slots.

Consider an initially full cache receiving κ new physical KV slots per step. For one-token decoding steps, $\kappa = L \times H$. Given a fixed, executable eviction order Q, assume $n\kappa \leq B_{\text{phys}}$ and no reordering or recall over n steps. Bulk eviction removes the first $n\kappa$ candidates at the outset, whereas ideal on-demand eviction (Lazy) removes only the κ candidates needed at each step. After admitting the new KVs at step j, cache occupancy is

$$N_j^{\text{Bulk}} = B_{\text{phys}} - (n-j)\kappa, \qquad N_j^{\text{Lazy}} = B_{\text{phys}}, \qquad j = 1, \ldots, n. \tag{2}$$

If d_j slots have been evicted cumulatively by step j, the budget constraint requires

$$B_{\text{phys}} + j\kappa - d_j \leq B_{\text{phys}} \quad \Longrightarrow \quad d_j \geq j\kappa. \tag{3}$$

On-demand eviction attains this lower bound. Thus, for a fixed eviction order, it releases only the capacity currently required and retains historical KVs for as long as the budget permits.

We further isolate the effect of eviction timing on continuous text from PG-19 (Rae et al., 2019). Using fixed eviction orders constructed by SnapKV and PyramidKV, we compare bulk and on-demand execution from the same full-budget cache, with eviction timing as the only varied factor. Figure 4 shows that on-demand execution remains closer to FullCache at all 15 measured checkpoints, reducing predictive KL by 77.5% and 66.7% on average under the SnapKV- and PyramidKV-derived orders, respectively. This leads to our second design requirement: *eviction should be planned periodically for efficiency but executed only when capacity is required*. Full experimental details are provided in Appendix B.

2.3 Eviction Should Not Imply Permanent Forgetting

On-demand eviction delays removal but cannot avoid it indefinitely under a fixed GPU budget. More importantly, a KV assessed as low-risk for removal at one point need not remain irrelevant as generation evolves. Prior work shows that one-shot compression can discard tokens needed later and that historical states can regain importance as attention shifts (Xu et al., 2025; Zhao et al., 2026; Zhang et al., 2026). Once such states are permanently deleted, subsequent rescoring cannot recover them. This suggests separating *GPU eviction* from *permanent forgetting*: evicted states should remain recoverable if their relevance later resurges.

A natural solution is to retrieve historical states when they become relevant again. However, if retrieved KVs are required by the current attention computation, CPU search and transfer can enter the decoding critical path. Recovery should thus be decoupled from the current decoding step. This leads to our third design requirement: *historical KVs should be recoverable without requiring current decoding to wait for their retrieval*. Section 3.3 realizes it through Asynchronous Recall.

3 LazyRecall: Decision-Risk-Driven Online KV Management

In this section, we present our method called **LazyRecall**, which addresses the three challenges identified in Section 2 through three corresponding mechanisms, i.e., (1) **Decision-Risk** for globally comparable KV selection, (2) **On-Demand Removal** for on-demand execution of eviction decisions, and (3) **Asynchronous Recall** for recovering historical states whose relevance resurges. As illustrated in Figure 5, LazyRecall uses periodic risk audits to construct a model-wide eviction order for on-demand execution during decoding. Asynchronously recalled candidates join subsequent audits, allowing the active cache to be updated under a fixed GPU KV budget B.

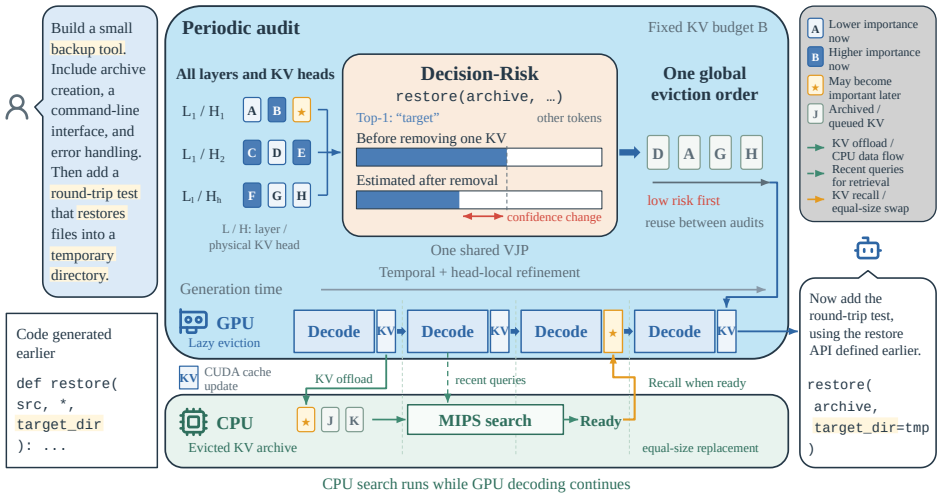


Figure 5: The LazyRecall framework. Periodic Decision-Risk audits construct a model-wide eviction order for incremental GPU execution, while asynchronous CPU retrieval supplies historical candidates for re-evaluation.

3.1 Decision-Risk: Final-Decision-Aware KV Scoring

The deletion risk in Equation (1) measures the change in predictive distributions after an actual removal. Scoring every candidate this way would require removing each KV independently and recomputing the downstream network, which is prohibitively expensive across all layers, heads, and historical positions. We instead derive the local attention perturbation analytically and estimate its effect on the final decision using a single shared backward pass.

Consider candidate i in physical KV head h of layer l, with value v_i. For query head r sharing this KV, let $\alpha_{r,i}$ denote its attention weight and $o_{l,r}$ the attention output. Removing i both subtracts its

weighted value and renormalizes the remaining attention weights. With the current query and all other KVs fixed, the exact local output change is

$$\Delta o_{l,r,i} := o_{l,r} - o_{l,r}^{(-i)} = \frac{\alpha_{r,i}}{1-\alpha_{r,i}}(v_i - o_{l,r}). \tag{4}$$

This perturbation lies in a head-specific representation space, so its magnitude alone does not determine its effect on the final prediction. To compare candidates across layers and heads, we map their local perturbations to the same output quantity: the confidence assigned to the current top-1 prediction. Given logits z, let y denote the currently most probable token, and define its margin against the aggregate of the remaining vocabulary as

$$y = \arg\max_j z_j, \qquad m = z_y - \log \sum_{j \neq y} e^{z_j}. \tag{5}$$

Let $p_y = \mathrm{softmax}(z)_y$. The margin satisfies

$$m = \log \frac{p_y}{1-p_y}, \qquad \sigma(m) = p_y,$$

where σ denotes the sigmoid function. Thus, a change in m directly represents a change in the model's confidence in its current top-1 prediction.

We next estimate how removing candidate i changes this common decision quantity. Holding y fixed, a first-order expansion of m with respect to the attention outputs gives

$$g_{l,r} = \frac{\partial m}{\partial o_{l,r}}, \qquad d_i = \sum_{r \mapsto h} g_{l,r}^\top \Delta o_{l,r,i}, \qquad m^{(-i)} \approx m - d_i. \tag{6}$$

Here, $g_{l,r}$ measures the downstream sensitivity of the common decision margin to the output of query head r in layer l. Because $\Delta o_{l,r,i} = o_{l,r} - o_{l,r}^{(-i)}$, the projected quantity d_i estimates the decrease in the margin caused by removing candidate i.

The sum covers all query heads that share physical KV head h in grouped-query attention (GQA) (Ainslie et al., 2023), since removing one physical KV jointly perturbs their attention outputs. A single shared vector–Jacobian product (VJP) computes $g_{l,r}$ for all layers and query heads. All candidates then reuse these gradients, and their projections and aggregation can be evaluated in parallel without rerunning the downstream network for each candidate.

Finally, we measure confidence changes in probability space by grouping the vocabulary into y and all remaining tokens. This yields the nonnegative *Decision-Risk* score

$$D_i = D_{\mathrm{KL}}[\mathrm{Ber}(\sigma(m)) \parallel \mathrm{Ber}(\sigma(m - d_i))], \tag{7}$$

where Ber denotes a Bernoulli distribution. A smaller D_i indicates a smaller estimated change in decision confidence. Evaluating every candidate against the same prediction event provides a common risk scale across layers, heads, and positions. The local attention perturbation is exact; the approximation concerns its downstream effect on the margin. Appendix C provides the derivation and approximation-error analysis under local smoothness assumptions.

3.2 On-Demand Removal: Plan Periodically, Evict On Demand

On-Demand Removal separates *eviction planning* from its actual *execution*. At each periodic audit, we construct a robust model-wide eviction queue for reuse over a nominal window of W decoding steps. The queue combines temporal risk aggregation with neighborhood protection while preserving the global allocation induced by Decision-Risk. During subsequent decoding, the queue is consumed lazily: KV states are evicted only when newly generated states require additional capacity. In short, On-Demand Removal follows a simple principle: *plan periodically, evict on demand*.

Robust Queue Construction. Because an eviction queue is reused across multiple decoding steps, a ranking computed from only the current query may become stale before its corresponding eviction is executed. We therefore aggregate Decision-Risk over a short window of recent queries. Let $\mathcal{S}$

denote the retained query set, and let $D_{s,i}$ denote the proxy risk score of candidate i under query s. We define the temporal risk of candidate i as

$$T_i = \max_{s \in \mathcal{S}} D_{s,i}. \tag{8}$$

Thus, a KV state remains protected if it is important to any recent query, reducing the sensitivity of the eviction order to the instantaneous query at the audit step. This aggregation does not require storing historical gradients or attention maps: all retained queries share the current audit VJP, while only their query-specific local deletion perturbations are recomputed. Moreover, the convexity of the risk function allows the temporal maximum to be evaluated using only the extrema of these perturbations, requiring at most two nonlinear risk evaluations per candidate (Appendix C).

To avoid isolated eviction within locally important regions, we apply neighborhood protection within each physical KV head. For candidate i, let $\mathcal{N}(i)$ denote its local neighborhood. We define

$$P_i = \max_{j \in \mathcal{N}(i)} T_j. \tag{9}$$

A candidate therefore receives a high pooled risk if either itself or a nearby state has recently been important. Directly sorting all candidates by P_i, however, may distort the model-wide allocation determined by the unpooled temporal risks T_i. We therefore separate *global allocation* from *local candidate selection*. Specifically, we first sort all candidates globally by the unpooled temporal risks T_i. This global ordering determines how many eviction slots are assigned to each layer and physical KV head, as well as the positions of these slots in the model-wide queue. Within each physical KV head, we then select the same number of candidates according to the pooled risks P_i and place them into its assigned queue slots in ascending pooled-risk order. Consequently, every prefix of the resulting queue preserves the layer/head eviction counts induced by the global risk ordering, while neighborhood pooling determines which local positions are safer to remove.

Lazy Execution. Once the global eviction queue is constructed, LazyRecall does not immediately remove all planned candidates. Instead, the queue is reused until the next audit and consumed only when incoming KVs actually require capacity. Suppose a decoding step introduces κ new physical KV slots after the active cache has reached its budget B. LazyRecall evicts only the next κ valid candidates from the queue before admitting the new states. Hence, scoring and queue construction are performed periodically, whereas eviction is executed on demand.

This execution policy keeps the active GPU cache at budget B after saturation while retaining historical states for as long as the capacity constraint permits. At the next scheduled audit, the remaining queue is discarded and rebuilt from the current cache state. If the queue no longer contains enough valid candidates to serve incoming states, LazyRecall triggers an early audit and constructs a new queue. Recalled candidates, discussed next, likewise invalidate the existing ordering and participate in the subsequent audit together with resident KVs.

3.3 Asynchronous Recall: Making Eviction Reversible

On-Demand Removal delays removal but cannot avoid it under a fixed GPU budget. Moreover, a KV that is low-risk when evicted may become relevant again as generation evolves. LazyRecall therefore makes eviction reversible by retaining evicted states in a CPU archive and allowing them to re-enter the active cache. Crucially, unlike retrieval-based attention, LazyRecall does not require retrieved KVs to complete the current attention computation. Instead, retrieval proposes historical candidates for future cache residency, allowing recovery to proceed asynchronously with GPU decoding.

Candidate Retrieval. LazyRecall maintains a CPU archive with a separate inner-product index for each layer–physical KV head pair. At periodic retrieval steps t, it averages the post-RoPE queries (Su et al., 2024) sharing physical KV head h in layer l to obtain $\bar{q}_{t,l,h}$. For archive $\mathcal{A}_{l,h}$, maximum inner-product search (MIPS) returns

$$\widehat{\mathcal{R}}_{t,l,h} = \mathrm{TopK}_{i \in \mathcal{A}_{l,h}} \left(\bar{q}_{t,l,h}^{\top} k_i \right), \tag{10}$$

where k_i is the archived key and TopK selects the K candidates with the largest inner products. These retrieval results are treated only as *candidates* for recovery rather than KVs required by the current attention computation.

Asynchronous Recovery. CPU search and CPU–GPU transfer run in the background and can overlap with GPU decoding between risk audits. Retrieved candidates are first placed in a GPU staging buffer separate from the active cache. At a subsequent risk audit, any ready candidates are evaluated alongside resident KVs; if none are ready, the audit proceeds with resident KVs alone. The staging buffer does not participate in decoding and is therefore excluded from the active-cache budget B, but its allocation is included in the peak GPU memory reported in Section 4.3.

Joint Risk Re-evaluation. A retrieval hit does not imply retention. Ready candidates participate in the same Decision-Risk audit together with resident KVs and are evaluated on the same model-wide risk scale. The same shared VJP evaluates both recalled and resident states, requiring no separate backward pass for recovery. Historical states whose relevance has increased can therefore regain GPU residency, whereas low-risk KVs, whether recalled or previously resident, will be evicted during the same audit. The resulting risk ordering then forms a new eviction queue for subsequent On-Demand Removal, closing the loop between eviction, recovery, and re-evaluation.

Implementation. We design dedicated CUDA kernels for risk computation, archive packing, cache compaction, and recall merging. Shared buffers and a separate transfer stream support asynchronous CPU–GPU execution. Implementation details and ablation studies are provided in Appendix E and Appendix D, respectively.

4 Experiments

We evaluate LazyRecall on long-form writing and mathematical reasoning from three perspectives: (1) generation quality under restricted active GPU KV budgets, (2) inference efficiency and GPU memory usage, and (3) the contributions of its individual components. We report representative end-to-end and efficiency results in the main text, with complete results, ablations, and sensitivity analyses provided in the appendices.

4.1 Experimental Setup

Hardware and Baselines. All experiments are conducted on a server equipped with two 10-core Intel Xeon Silver CPUs and a single NVIDIA RTX A6000 GPU with 48 GB of memory. We compare LazyRecall with R-KV (Cai et al., 2026), SCOPE (Wu et al., 2025a), and BeaconKV (Kim et al., 2026), using uncompressed FullCache as a reference. These baselines select KVs using importance and redundancy, manage prefill and decoding caches separately, or use representative historical queries to guide selection, respectively. All compression methods use the same generation parameters and active GPU KV budget, whereas FullCache retains the complete KV history.

Long-Form Writing. We use the Short and Long splits of LongGenBench (Wu et al., 2025b), which require models to satisfy position-specific, range-based, and periodic constraints throughout extended outputs. We evaluate Llama-3.1-8B-Instruct and Mistral-7B-Instruct, reporting Completion, Once Accuracy, and Range Accuracy.

Mathematical Reasoning. We evaluate DeepSeek-R1-Distill-Llama-8B and Qwen3.5-9B on AIME I and AIME II from AIME 2024. We generate 8 responses per problem and report final-answer accuracy over all responses. All methods use the same scoring procedure.

4.2 End-to-End Generation Quality

We first compare end-to-end generation quality under the lower active GPU KV-cache budget of $B = 2{,}048$. We evaluate Llama-3.1-8B-Instruct on LongGenBench-Short and DeepSeek-R1-Distill-Llama-8B on AIME 2024. Table 1 reports representative results, while complete results across models and budgets are provided in Appendix A.

Despite the restricted active KV budget, LazyRecall preserves strong generation quality across both extended instruction following and mathematical reasoning. On LongGenBench-Short, it achieves the highest Once and Range accuracy among the evaluated methods, while remaining competitive in

Table 1: Generation quality (%, ↑) at $B = 2{,}048$ on LongGenBench-Short with Llama-3.1-8B-Instruct and AIME 2024 with DeepSeek-R1-Distill-Llama-8B. Dark green highlights the highest value in each column, including the uncompressed FullCache reference. Light green highlights a result that is the best among compression methods but not the highest in the column.

	LongGenBench-Short Avg. output: 8,000 tokens			**AIME 2024** Avg. output: 13,235 tokens	
Method	Completion	Once Acc.	Range Acc.	AIME I Acc.	AIME II Acc.
FullCache	66.09	30.57	40.61	**51.67**	**53.33**
R-KV	64.47	34.78	41.05	47.50	42.50
BeaconKV	**69.81**	33.00	42.91	43.33	46.67
SCOPE	62.48	29.55	39.84	40.00	30.83
LazyRecall	65.26	**36.32**	**43.50**	47.50	**53.33**

Table 2: Inference efficiency and GPU memory usage.

Method	Throughput (tokens/s) ↑	Time/token (ms) ↓	Peak allocated (GiB) ↓	Above model baseline (GiB) ↓
FullCache	**24.620**	**40.617**	16.273	1.316
R-KV	21.401	46.726	16.183	1.226
SCOPE	12.748	78.445	16.355	1.397
BeaconKV	19.758	50.613	15.445	0.487
LazyRecall	23.662	42.262	**15.390**	**0.433**

completion rate. On AIME 2024-I, LazyRecall ties R-KV for the best accuracy among the compression methods; on AIME 2024-II, it matches FullCache and outperforms all compression baselines. Complete results in Appendix A further show consistent quality preservation across additional models, longer generations, and different KV budgets. Ablation studies and hyperparameter sensitivity analyses are provided in Appendix D.

4.3 Inference Efficiency and Memory Usage

Table 2 compares generation throughput, time per output token, and peak allocated GPU memory on Llama-3.1-8B-Instruct. All methods generate 10,000 output tokens from the same input under identical generation settings. The compression methods use an active GPU KV budget of $B = 2{,}048$, whereas FullCache retains the complete KV history.

Despite online risk auditing and historical recall, LazyRecall achieves 23.662 tokens/s, the highest throughput among the evaluated compression methods, while retaining 96.1% of FullCache throughput with only approximately 4.0% higher time per output token. LazyRecall also has the lowest peak GPU memory allocation at 15.390 GiB. Relative to FullCache, total peak allocation decreases by 5.4%, while allocation above the approximately 14.957 GiB model baseline decreases from 1.316 to 0.433 GiB, a 67.1% reduction. The smaller reduction in total GPU memory reflects the large, fixed model footprint. Overall, these results show that LazyRecall substantially reduces GPU allocation above the model baseline while introducing little additional decoding overhead.

5 Conclusion

We presented *LazyRecall*, a decision-risk-driven framework for online KV-cache management that jointly addresses KV selection, eviction timing, and historical recovery. *Decision-Risk* estimates the effect of KV removal on the final model decision to enable model-wide selection on a common risk scale. *On-Demand Removal* decouples periodic eviction planning from on-demand execution, while *Asynchronous Recall* keeps evicted states recoverable without blocking current decoding. Experiments on long-form writing and mathematical reasoning show that LazyRecall maintains generation quality under substantially reduced active GPU KV budgets while introducing little additional decoding overhead. These results demonstrate the effectiveness of combining risk-aware selection, demand-driven eviction, and recoverable historical states for online KV-cache management.

AI USE STATEMENT

Generative AI tools were used to assist with literature discovery, translation, and language editing. All AI-assisted content was reviewed and verified by the authors, who take full responsibility for the final manuscript.

REFERENCES

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. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 4895–4901, 2023.

Yongqi An, Chang Lu, Kuan Zhu, Tao Yu, Chaoyang Zhao, Hong Wu, Ming Tang, and Jinqiao Wang. Rest-kv: Robust kv cache eviction with layer-wise output reconstruction and spatial-temporal smoothing. *arXiv preprint arXiv:2605.08840*, 2026.

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. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 3119–3137, 2024.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–43, 2018.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

Zefan Cai, Wen Xiao, Hanshi Sun, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, et al. R-kv: Redundancy-aware kv cache compression for reasoning models. *Advances in neural information processing systems*, 38:60980–61005, 2026.

Chi-Chih Chang, Wei-Cheng Lin, Chien-Yu Lin, Chong-Yan Chen, Yu-Fang Hu, Pei-Shuo Wang, Ning-Chi Huang, Luis Ceze, Mohamed S Abdelfattah, and Kai-Chiang Wu. Palu: Compressing kv-cache with low-rank projection. *arXiv preprint arXiv:2407.21118*, 2024.

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. In *International Conference on Learning Representations*, volume 2025, pp. 44169–44190, 2025.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pp. 35549–35562, 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. *Advances in Neural Information Processing Systems*, 38:113152–113188, 2026.

Yu Fu, Zefan Cai, Abedelkadir Asi, Wayne Xiong, Yue Dong, and Wen Xiao. Not all heads matter: A head-level kv cache compression method with integrated retrieval and reasoning. In *International Conference on Learning Representations*, volume 2025, pp. 99269–99290, 2025.

Yuzhe Gu, Xiyu Liang, Jiaojiao Zhao, and Enmao Diao. Obcache: Optimal brain kv cache pruning for efficient long-context llm inference. *arXiv preprint arXiv:2510.07651*, 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: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Janghyeon Kim, Minsoo Kim, Kyuhong Shim, and Jungwook Choi. Beaconkv: Key-value cache compression guided by beacon queries for efficient large reasoning model inference. *arXiv preprint arXiv:2609.04971*, 2026.

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.

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.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Akide Liu, Jing Liu, Zizheng Pan, Yefei He, Gholamreza Haffari, and Bohan Zhuang. Minicache: Kv cache compression in depth dimension for large language models. *Advances in Neural Information Processing Systems*, 37:139997–140031, 2024a.

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. *Advances in Neural Information Processing Systems*, 38:54358–54385, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024b.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36:52342–52364, 2023.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024c.

Tho Mai and Joo-Young Kim. Reformulating kv cache eviction problem for long-context llm inference. *arXiv preprint arXiv:2605.07234*, 2026.

Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

Yiqun Shen, Song Yuan, Zhengze Zhang, Xiaoliang Wang, Daxin Jiang, and Cam-Tu Nguyen. LAVa: Layer-wise KV cache eviction with dynamic budget allocation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 13672–13692. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.737. URL https://aclanthology.org/2025.findings-emnlp.737/.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Jialong Wu, Zhenglin Wang, Linhai Zhang, Yilong Lai, Yulan He, and Deyu Zhou. Scope: Optimizing key-value cache compression in long-context generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10775–10790, 2025a.

Wenbo Wu, Qingyi Si, Xiurui Pan, Ye Wang, and Jie Zhang. Louiskv: Efficient kv cache retrieval for long input-output sequences. In *International Conference on Learning Representations*, volume 2026, pp. 65401–65421, 2026.

Yuhao Wu, Ming Shan Hee, Zhiqiang Hu, and Roy Ka-Wei Lee. Longgenbench: Benchmarking long-form generation in long context llms. In *International Conference on Learning Representations*, volume 2025, pp. 6851–6872, 2025b.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024.

Fangyuan Xu, Tanya Goyal, and Eunsol Choi. Refreshkv: Updating small kv cache during long-form generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 24878–24893, 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.

Haoyue Zhang, Hualei Zhang, Xiaosong Ma, Jie Zhang, and Song Guo. Lazyeviction: Lagged kv eviction with attention pattern observation for efficient long reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 36335–36352, 2026.

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.

Yi Zhao, Yajuan Peng, Nguyen Cam-Tu, Zuchao Li, Xiaoliang Wang, Hai Zhao, and Xiaoming Fu. Smallkv: Small model assisted compensation of kv cache compression for efficient llm inference. *Advances in Neural Information Processing Systems*, 38:48637–48665, 2026.

A SUPPLEMENTARY EXPERIMENTS

A.1 BENCHMARK AND BASELINE SELECTION

Our end-to-end experiments cover long-form writing and mathematical reasoning under multiple active GPU KV-cache budgets. The Short and Long splits of LongGenBench evaluate instruction following throughout extended generation (Wu et al., 2025b). Completion rate is the fraction of required subtasks with valid generated outputs, counted by unique subtask identifiers. Once accuracy measures whether requirements at individual specified positions are satisfied, whereas Range accuracy checks requirements at each position within a specified contiguous range. Each accuracy is the number of passed checks divided by the number of evaluated checks in that category. Checks are performed only at positions with corresponding generated outputs; missing outputs are reflected in completion rate. AIME I and AIME II from AIME 2024 assess multi-step mathematical reasoning, with accuracy determined by matching the extracted final integer answer to the ground truth.

We use R-KV, SCOPE, and BeaconKV as compression baselines and uncompressed FullCache as a reference (Cai et al., 2026; Wu et al., 2025a; Kim et al., 2026). These compression baselines support online KV management during generation and can be compared under a common budget constraint. We retain their open-source default method hyperparameters, except for changes required to align budgets, generation conditions, and model interfaces.

The native cache-management schedules of methods such as SnapKV and ReST-KV differ from the fixed-budget online setting considered in our end-to-end evaluation (Li et al., 2024; An et al., 2026). In particular, matching their initially retained cache sizes does not guarantee the same active GPU KV budget throughout subsequent generation, while enforcing such a constraint would require modifying their native management procedures. We therefore do not include them in the direct end-to-end system comparison. Instead, in Appendix B.3, we evaluate their scoring and selection strategies under a common compression procedure using predictive KL divergence relative to FullCache. This controlled comparison isolates the quality of KV selection from differences in cache-management and eviction schedules.

A.2 COMPLETE END-TO-END RESULTS

This section provides the complete end-to-end results underlying and extending those reported in Section 4. We report LongGenBench-Short, LongGenBench-Long, and AIME 2024 separately because they use different models, generation lengths, KV-cache budgets, and evaluation metrics. Within each experimental configuration, all methods use the same model, inputs, prompt templates, and decoding parameters. FullCache retains the complete KV history and serves as the uncompressed reference.

R-KV, SCOPE, and BeaconKV are reproduced from their publicly available codebases using the default method-specific hyperparameters provided by the original implementations. Except for the modifications required to support the evaluated models and enforce a common active GPU KV budget, we do not alter their scoring, compression, or cache-management strategies. For LongGenBench, we use a temperature of 0.8, top-$p = 0.95$, and maximum generation lengths of 8,000 and 16,000 tokens for the Short and Long splits, respectively. For AIME, we use a temperature of 0.6, top-$p = 0.95$, and generate eight responses per problem. The budget B is measured in token-equivalent units and corresponds to $B_{\text{phys}} = B \times L \times H$ physical KV slots.

LongGenBench-Short. Table A1 reports the complete results on LongGenBench-Short. We evaluate Llama-3.1-8B-Instruct and Mistral-7B-Instruct under $B = 2{,}048$ and $B = 4{,}096$. The average output length is approximately 8,000 tokens. Completion measures the fraction of required subtasks for which valid outputs are produced, while Once Accuracy and Range Accuracy measure whether the model satisfies constraints at specified positions and over specified contiguous ranges, respectively.

On Llama-3.1-8B-Instruct, LazyRecall achieves the highest Once Accuracy and Range Accuracy under both budgets, while BeaconKV achieves the highest Completion. On Mistral-7B-Instruct, LazyRecall achieves the highest Range Accuracy at $B = 2{,}048$ and the best result on all three met-

rics at $B = 4{,}096$. Overall, LazyRecall obtains the best or tied-best result among the compression methods in 8 of the 12 model–budget–metric configurations in the Short setting.

Table A1: Complete results on LongGenBench-Short, with an average output length of approximately 8,000 tokens. Results are reported under active GPU KV-cache budgets of $B = 2{,}048$ and $B = 4{,}096$, measured in token-equivalent units.

		$B = 2{,}048$			$B = 4{,}096$		
Model	Method	Completion	Once Acc.	Range Acc.	Completion	Once Acc.	Range Acc.
Llama-3.1-8B-Instruct	FullCache	66.09	30.57	40.61	66.09	30.57	40.61
	R-KV	64.47	34.78	41.05	65.51	34.48	41.92
	BeaconKV	**69.81**	33.00	42.91	**68.25**	33.95	43.12
	SCOPE	62.48	29.55	39.84	63.98	31.46	39.62
	LazyRecall	65.26	**36.32**	**43.50**	65.71	**36.48**	**44.02**
Mistral-7B-Instruct	FullCache	51.24	22.79	23.89	51.24	22.79	23.89
	R-KV	53.57	**32.73**	25.77	52.14	29.88	25.37
	BeaconKV	**55.64**	26.60	27.64	53.52	27.62	26.83
	SCOPE	49.97	22.76	21.14	54.46	24.55	25.61
	LazyRecall	53.76	28.90	**28.86**	**54.57**	**29.92**	**27.64**

LongGenBench-Long. Table A2 reports the corresponding results on LongGenBench-Long. This setting extends the average output length to approximately 16,000 tokens and uses budgets of $B = 4{,}096$ and $B = 8{,}192$, testing whether each method can preserve instruction-following performance over longer generation trajectories.

On Llama-3.1-8B-Instruct, LazyRecall achieves the highest Once Accuracy at $B = 4{,}096$ and ties R-KV for the highest Range Accuracy. At $B = 8{,}192$, it achieves the highest Completion and Once Accuracy. On Mistral-7B-Instruct, LazyRecall achieves the highest Once Accuracy and Range Accuracy among the compression methods under both budgets, and also achieves their highest Completion at $B = 8{,}192$. Overall, LazyRecall obtains the best or tied-best result among the compression methods in 9 of the 12 configurations in the Long setting, showing that its advantage extends to longer generation trajectories.

Table A2: Complete results on LongGenBench-Long, with an average output length of approximately 16,000 tokens. Results are reported under active GPU KV-cache budgets of $B = 4{,}096$ and $B = 8{,}192$, measured in token-equivalent units.

		$B = 4{,}096$			$B = 8{,}192$		
Model	Method	Completion	Once Acc.	Range Acc.	Completion	Once Acc.	Range Acc.
Llama-3.1-8B-Instruct	FullCache	54.70	40.59	23.76	54.70	40.59	23.76
	R-KV	**56.78**	46.65	26.64	53.90	44.98	28.75
	BeaconKV	53.55	43.30	23.25	53.97	44.58	26.43
	SCOPE	53.64	45.93	25.58	54.30	47.13	**30.14**
	LazyRecall	54.11	**47.77**	**26.64**	**54.80**	**48.58**	28.96
Mistral-7B-Instruct	FullCache	24.56	22.68	15.83	24.56	22.68	**15.83**
	R-KV	28.93	22.97	11.42	29.16	22.25	13.32
	BeaconKV	**30.08**	25.36	14.74	28.96	26.12	14.91
	SCOPE	19.43	19.61	12.88	27.46	23.44	14.59
	LazyRecall	29.83	**26.21**	**16.70**	**29.47**	**28.21**	15.11

AIME 2024. Table A3 reports the complete results on AIME 2024-I and AIME 2024-II. We evaluate DeepSeek-R1-Distill-Llama-8B and the thinking-enabled Qwen3.5-9B. We generate eight responses per problem and compute accuracy over all generated responses. We extract the final integer answer using the same procedure for all methods; a response is counted as incorrect if no valid integer can be extracted or if the extracted answer does not match the ground truth.

On DeepSeek-R1-Distill-Llama-8B, LazyRecall ties R-KV as the best compression method on AIME 2024-I at $B = 2{,}048$. It achieves the highest accuracy among the compression methods under both budgets on AIME 2024-II, matching FullCache at $B = 2{,}048$. On Qwen3.5-9B, LazyRecall outperforms the other compression methods in all four dataset–budget configurations. On AIME 2024-II at $B = 8{,}192$, it reaches 67.50% accuracy, compared with 66.67% for FullCache. LazyRecall also leads the compression methods on AIME 2024-I with DeepSeek-R1-Distill-Llama-8B at $B = 4{,}096$. Thus, it achieves the best or tied-best result among the compression methods in all eight AIME configurations.

Table A3: Complete accuracy results (%, ↑) on AIME 2024-I and AIME 2024-II. DeepSeek-R1-Distill-Llama-8B generates approximately 13,235 tokens per problem on average, whereas the thinking-enabled Qwen3.5-9B generates approximately 24,557 tokens.

Dataset	Method	DeepSeek-R1-Distill-Llama-8B Avg. output: 13,235 tokens		Qwen3.5-9B (Thinking) Avg. output: 24,557 tokens	
		$B = 2{,}048$ Accuracy	$B = 4{,}096$ Accuracy	$B = 4{,}096$ Accuracy	$B = 8{,}192$ Accuracy
AIME 2024-I	FullCache	**51.67**	**51.67**	**73.33**	**73.33**
	R-KV	47.50	46.67	32.50	48.33
	BeaconKV	43.33	47.50	33.33	51.67
	SCOPE	40.00	48.33	23.33	45.83
	LazyRecall	47.50	50.83	40.83	72.50
AIME 2024-II	FullCache	53.33	**53.33**	**66.67**	66.67
	R-KV	42.50	46.67	35.83	53.33
	BeaconKV	46.67	51.67	38.33	51.67
	SCOPE	30.83	46.67	29.17	53.33
	LazyRecall	**53.33**	52.50	39.17	**67.50**

Overall results. Across the evaluated settings, LazyRecall consistently preserves strong generation quality under restricted active GPU KV budgets, achieving the best or tied-best result among the compression methods in 25 of the 32 reported configurations.

B Evaluation Metrics and Controlled Comparisons

Sections 2.1 and 2.2 both use KL divergence between full-vocabulary predictive distributions, but with different references. Section 2.1 compares predictions before and after independently removing a single KV from a fixed decoding state, measuring its instantaneous deletion risk. Section 2.2 compares a compressed-cache branch with an uncompressed FullCache branch at the same continuation step, measuring the effect of cache management on subsequent predictions. The former uses the pre-removal cache as its reference; the latter uses FullCache.

B.1 Single-KV Deletion Risk and Scoring Diagnostics

At a fixed decoding checkpoint, let p be the next-token predictive distribution before removal. For a candidate physical KV i, we independently remove it from the same cache state and recompute the model output, obtaining $p^{(-i)}$. Its measured single-KV deletion risk is

$$\mathcal{K}_i = D_{\mathrm{KL}}\left(p \,\middle\|\, p^{(-i)}\right) = \sum_{v \in \mathcal{V}} p(v) \log \frac{p(v)}{p^{(-i)}(v)}, \tag{11}$$

where $\mathcal{V}$ is the full vocabulary. We restore the original cache before evaluating the next candidate. Thus, $\mathcal{K}_i$ is obtained by recomputation after actual removal, rather than from the *Decision-Risk* approximation. We use it to evaluate the scoring quality of LAVa, OBCache, and base *Decision-Risk*.

Within-Score Risk Dispersion. We first ask whether KVs assigned similar scores have similar measured deletion risks. At the same checkpoint, each method forms narrow score groups in low-, middle-, and high-score regions using only its own scores. Measured KL is not used to form the groups. For a group G, let $\overline{\mathcal{K}}_G$ be its mean measured risk. Its coefficient of variation is

$$\mathrm{CV}_G = \frac{\mathrm{Std}_{i \in G}(\mathcal{K}_i)}{\overline{\mathcal{K}}_G}, \tag{12}$$

where Std denotes the sample standard deviation. A lower CV indicates that similarly scored candidates have more consistent removal effects.

Risk-Ranking Agreement. We next compare method scores with measured single-KV deletion risks on a predetermined common set of candidates. All methods are evaluated on the same candidates. After orienting all scores so that larger values indicate higher predicted deletion risk, we compute Spearman's rank correlation between a method's score s_i and $\mathcal{K}_i$:

$$\rho = \mathrm{Corr}(\mathrm{rank}(s_i), \mathrm{rank}(\mathcal{K}_i)) \,. \tag{13}$$

A higher ρ indicates closer agreement with the ranking induced by independently measured deletion risks.

Joint-Deletion Risk Ratio. Finally, we test the consequence of selecting a set of KVs for removal. This diagnostic uses a fixed, small candidate pool $\mathcal{U}$. Given a deletion count k, method M selects the k candidates with the lowest predicted deletion risk from $\mathcal{U}$, forming S_M. We remove the selected KVs jointly and recompute the predictive KL, denoted by $\mathcal{K}(S_M)$. We also enumerate all subsets of size k within the same pool and take the subset with the smallest joint-deletion KL as a restricted oracle:

$$S_\star = \underset{S \subseteq \mathcal{U},\ |S|=k}{\arg\min}\ \mathcal{K}(S), \qquad \mathrm{RM}_M = \frac{\mathcal{K}(S_M)}{\mathcal{K}(S_\star)}. \tag{14}$$

A ratio closer to 1 indicates that the method-selected set is closer to the best joint-deletion set within this candidate pool. The metric uses the KL measured after joint removal, not a sum of single-KV risks; the restricted oracle is not a global optimum over the full cache.

These diagnostics examine within-score consistency, single-KV risk ordering, and the consequence of selecting multiple KVs for removal. They do not directly measure end-to-end generation accuracy.

B.2 Predictive KL Relative to FullCache

To assess how cache changes affect subsequent predictions, we select ten sufficiently long continuous text segments from PG-19 (Rae et al., 2019). For each segment, the first 64 tokens form a protected prompt, the following 512 tokens establish a full-budget cache, and the next 150 tokens form the evaluation continuation. All branches consume the same continuation tokens, but independently execute forward passes and update their own KV caches. They therefore predict at the same text positions while retaining the cache-state differences caused by their respective eviction decisions.

Let p_t^{Full} denote the next-token distribution of the FullCache branch at continuation step t, and let p_t^M denote that of a compressed branch M. We measure

$$\mathcal{D}_t^M = D_{\mathrm{KL}}\big(p_t^{\mathrm{Full}} \,\big\|\, p_t^M\big) \,. \tag{15}$$

FullCache has zero divergence from itself by definition. A smaller $\mathcal{D}_t^M$ indicates that the compressed branch's prediction is closer to the uncompressed prediction. Unlike Equation equation 11, this measure compares two cache branches at the same continuation step, rather than predictions before and after removing one KV from a fixed cache.

Section 2.2 uses this measure to isolate eviction timing. We construct fixed eviction orders using the selection strategies of SnapKV and PyramidKV. For each order, Bulk and Lazy start from the same full-budget cache and receive the same continuation; the only experimentally varied factor is when the planned removals are executed. Bulk evicts the planned KVs in advance, whereas Lazy follows the same order only as new KVs require capacity. Neither branch reorders candidates or performs recall during the evaluation window.

The main-text curves report 15 checkpoints, sampled every 10 steps from step 10 through step 150. Each plotted KL value is the mean over the ten text segments at that step. After equally averaging the plotted checkpoints, Lazy reduces mean predictive KL relative to Bulk by 77.5% with the SnapKV-derived order and 66.7% with the PyramidKV-derived order. Lazy has lower KL at all 15 checkpoints under both orders. Thus, with the selection order held fixed, on-demand eviction keeps predictions closer to FullCache during cache refill. This result does not by itself establish an improvement in end-to-end task accuracy.

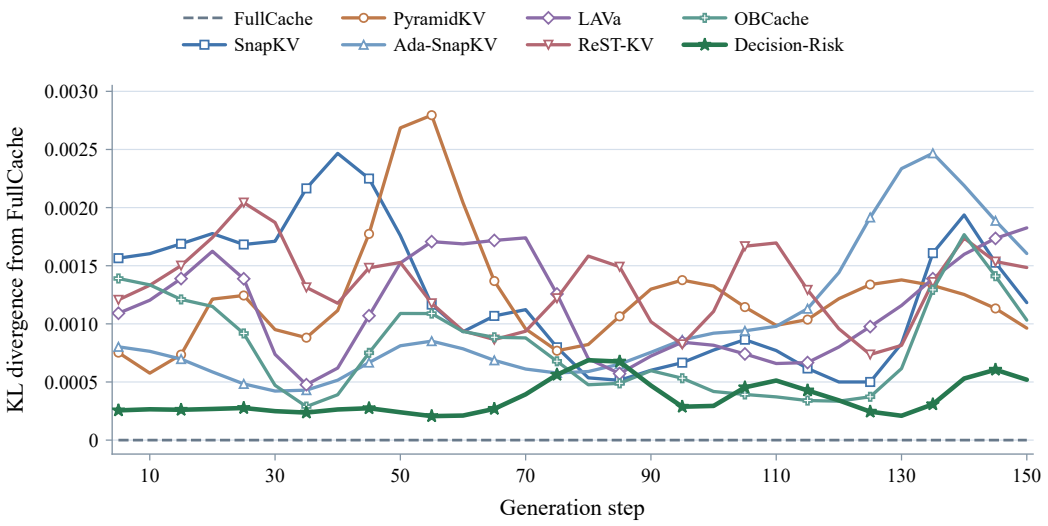


Figure B1: Predictive KL divergence relative to FullCache on PG-19 continuations. Each point averages ten text segments; lower is better. FullCache is the zero reference.

B.3 Controlled Comparison of KV Selection Methods

The native cache-management procedures of some KV-compression methods differ, so matching only their initial retained capacities does not establish an end-to-end comparison under identical online budgets. We therefore use the PG-19 continuations and predictive KL defined above to examine FullCache, SnapKV, PyramidKV, Ada-SnapKV, LAVa, ReST-KV, OBCache, and *Decision-Risk*. FullCache is not compressed and serves solely as the zero-KL reference.

For each text segment, the compression methods evict KV capacity equivalent to 150 tokens at the start of the observation window, then process the same 150-token continuation. The scoring interval is set to 150 steps; otherwise, each method retains the scoring and selection rules of its original implementation. At each measurement step, we compute its predictive KL relative to FullCache and average the results across the ten text segments. Each point in Figure B1 therefore represents a multi-segment mean at one continuation step, rather than the KL of an individual example.

This comparison measures how the information retained by each method affects subsequent predictions under the stated experimental conditions. It does not compare the runtime of the original systems, nor does the *Decision-Risk* curve include LazyRecall's asynchronous CPU recall. Figure B1 reports 30 checkpoints, sampled every five steps from step 5 through step 150. Among the compression methods, *Decision-Risk* has the lowest predictive KL at 24 of these checkpoints and the lowest mean KL across all 30.

This comparison is limited to methods with publicly available implementations whose scoring and selection rules can be evaluated within the common compression procedure. We do not report adaptations of methods without released implementations, as their behavior under this procedure cannot be reliably verified.

C Derivation and Extensions of Decision-Risk

C.1 Local Deletion Perturbation and Shared Gradients

For the current query, let $o_r = \sum_j \alpha_{r,j} v_j$ denote the attention output. With the query and all other KVs fixed, removing candidate i renormalizes the remaining weights:

$$o_r^{(-i)} = \frac{o_r - \alpha_{r,i} v_i}{1 - \alpha_{r,i}}, \qquad \Delta o_{r,i} := o_r - o_r^{(-i)} = \frac{\alpha_{r,i}}{1 - \alpha_{r,i}} (v_i - o_r). \tag{16}$$

This identity requires at least one visible KV to remain after removal. It accounts for both value removal and renormalization and is exact locally, rather than an approximation of the downstream network response.

Holding the currently most probable token $y = \arg\max_j z_j$ fixed, define

$$m = z_y - \log \sum_{j \neq y} e^{z_j}, \qquad \sigma(m) = \frac{e^{z_y}}{\sum_j e^{z_j}} = p_y. \tag{17}$$

A single shared VJP computes $g_{l,r} = \partial m / \partial o_{l,r}$ across layers and query heads (Baydin et al., 2018). For a candidate in physical KV head h of layer l, the first-order margin decrease is

$$d_i = \sum_{r \in \mathcal{R}(h)} g_{l,r}^\top \Delta o_{l,r,i}, \qquad \widehat{m}_i = m - d_i, \tag{18}$$

where $\mathcal{R}(h)$ contains all query heads sharing that physical KV head. All candidates reuse the same backward pass. Signed GQA contributions are summed before conversion to nonnegative risk.

C.2 Risk Expression and Approximation Error

Let $\mathrm{sp}(x) = \log(1 + e^x)$. Expanding the Bernoulli KL gives

$$\begin{aligned} D_i &= D_{\mathrm{KL}}[\mathrm{Ber}(\sigma(m)) \| \mathrm{Ber}(\sigma(m - d_i))] \\ &= \sigma(m) d_i - \mathrm{sp}(m) + \mathrm{sp}(m - d_i) \equiv \Phi_m(d_i). \end{aligned} \tag{19}$$

For the first-order margin estimate used in LazyRecall, the Bernoulli KL is evaluated without further approximation; only the post-removal margin is approximated. Convexity of softplus gives $\Phi_m(d) \geq 0$. The implementation uses stable softplus evaluation and safeguards the denominator $1 - \alpha$ near the probability boundary.

Conditional Error Bound for the First-Order Approximation. Concatenate the affected heads' outputs and perturbations into O_i and ΔO_i. With the checkpoint's other inputs and caches fixed, let f_i map this interface to the final margin for the fixed token y, so that $m = f_i(O_i)$ and $m_i^{\mathrm{exact}} = f_i(O_i - \Delta O_i)$. Assume f_i is twice continuously differentiable in a neighborhood of the line segment between these endpoints, with Hessian operator norm bounded by M_i along the segment. Taylor's integral remainder yields

$$\begin{aligned} m_i^{\mathrm{exact}} - \widehat{m}_i &= \int_0^1 (1 - t) \Delta O_i^\top \nabla^2 f_i(O_i - t \Delta O_i) \Delta O_i \, dt, \\ |m_i^{\mathrm{exact}} - \widehat{m}_i| &\leq \tfrac{1}{2} M_i \|\Delta O_i\|_2^2. \end{aligned} \tag{20}$$

For $\Psi_m(b) = D_{\mathrm{KL}}[\mathrm{Ber}(\sigma(m)) \| \mathrm{Ber}(\sigma(b))]$, we have $\Psi_m'(b) = \sigma(b) - \sigma(m)$, whose absolute value is at most one. Writing the exact post-removal binary risk as $R_i = \Psi_m(m_i^{\mathrm{exact}})$ gives

$$|D_i - R_i| \leq \tfrac{1}{2} M_i \|\Delta O_i\|_2^2. \tag{21}$$

This bound supports the first-order approximation under small perturbations and controlled curvature; it does not presume small errors for all candidates. The scoring diagnostics in Section 2.1 examine correspondence between approximate scores and measured deletion risks. Any clipping of the local perturbation for numerical stability introduces an additional clipping error.

Relationship to Vocabulary-Level KL. Let p and $q = p^{(-i)}$ be the actual pre- and post-removal predictions, and let $\widetilde{p}$ and $\widetilde{q}$ be their renormalized distributions over tokens other than y. Separating the term for y from the remaining terms gives

$$\begin{aligned} D_{\mathrm{KL}}(p\|q) = D_{\mathrm{KL}}[\mathrm{Ber}(p_y)\|\mathrm{Ber}(q_y)] \\ + (1-p_y)D_{\mathrm{KL}}(\widetilde{p}\|\widetilde{q}). \end{aligned} \tag{22}$$

Thus, the exact binary risk is no greater than the vocabulary-level KL. Decision-Risk targets the first term using a first-order margin estimate. This inexpensive score does not capture all redistribution among the other tokens, and its approximate value does not unconditionally satisfy $D_i \le \mathcal{K}_i$.

C.3 Endpoint Evaluation for Temporal Aggregation

For the first-order top-1 score used in LazyRecall, candidate perturbations $d_{s,i}$ for historical queries s share the current audit's margin and gradients. Let $d_i^- = \min_s d_{s,i}$ and $d_i^+ = \max_s d_{s,i}$. Since

$$\Phi_m''(d) = \sigma(m-d)(1-\sigma(m-d)) \ge 0, \tag{23}$$

the risk at any point in the interval is bounded by the larger endpoint risk. Both endpoints occur in the historical-query set, so

$$\max_s \Phi_m(d_{s,i}) = \max\{\Phi_m(d_i^-), \Phi_m(d_i^+)\}. \tag{24}$$

Temporal aggregation therefore requires only the perturbation extrema for each candidate, not all historical risk values. This identity applies to the temporal surrogate with fixed current gradients, not to a replay of historical output losses.

C.4 Decision Supports and Higher-Order Approximations

LazyRecall fixes the most probable token y under the pre-removal prediction and estimates how KV removal changes its decision confidence. This scalar margin allows all candidates to share a single VJP, but does not capture probability redistribution among other plausible outputs. We consider two extensions of this scoring rule: enlarging the decision support and increasing the approximation order of the downstream margin change.

Expanded Decision Supports. Following the support-selection rule of nucleus (top-p) sampling, we select the smallest set S_τ of high-probability tokens whose cumulative probability under the pre-removal distribution p reaches at least τ. The set remains fixed when evaluating different KV candidates. For any fixed support S, we preserve the probability of each token in S and aggregate all remaining tokens into one category:

$$\Pi_S(p) = \big((p_j)_{j\in S}, p_{\bar{S}}\big), \qquad p_{\bar{S}} = 1 - \sum_{j\in S} p_j. \tag{25}$$

Let $q = p^{(-i)}$ be the predictive distribution recomputed after removing candidate i. The exact support-based deletion risk is

$$\mathcal{R}_i(S) = D_{\mathrm{KL}}(\Pi_S(p)\|\Pi_S(q))\,. \tag{26}$$

It measures both the change in the support's total probability relative to the outside category and the redistribution among tokens within the support. When $S = \{y\}$, the latter vanishes and $\mathcal{R}_i(S)$ reduces to the exact binary risk discussed in Section C.2. Redistribution within the outside category is still omitted, so $\mathcal{R}_i(S)$ is not the full-vocabulary deletion KL. Selecting S_τ by the nucleus rule also does not make this quantity the KL divergence between truncated, renormalized top-p sampling distributions.

Equation equation 26 uses an actual post-removal prediction and is not itself an efficient score. To obtain a first-order approximation, for each $j \in S$ define its margin against the aggregate outside category as

$$m_j(S) = z_j - \log \sum_{v\notin S} e^{z_v}. \tag{27}$$

For candidate i in physical KV head h of layer l, we use the local perturbation from Section C.1 to estimate the post-removal margin:

$$\widehat{m}_{j,i}(S) = m_j(S) - \sum_{r \in \mathcal{R}(h)} \left(\frac{\partial m_j(S)}{\partial o_{l,r}} \right)^{\top} \Delta o_{l,r,i}. \tag{28}$$

Applying softmax to the estimated margins together with a zero logit for the outside category gives an approximate post-removal distribution $\widehat{\Pi}_{S,i}$ and score

$$\widehat{D}_i(S) = D_{\mathrm{KL}}\left(\Pi_S(p) \middle\| \widehat{\Pi}_{S,i} \right). \tag{29}$$

For $S = \{y\}$, this construction recovers the first-order Decision-Risk score D_i in Equation equation 19. Under the same reverse-mode implementation, a support containing $s = |S|$ tokens requires sensitivities for s margins and their projections onto the candidate perturbations, rather than one shared scalar-margin VJP. Computing these sensitivities sequentially limits the number of simultaneously stored gradients, whereas retaining them together increases temporary memory use; either way, the additional gradient and projection work grows with s.

Higher-Order Downstream Approximations. To isolate the effect of first-order linearization, these variants keep $S = \{y\}$ fixed and change only the approximation of the post-removal margin. Using f_i, O_i, and ΔO_i from Section C.2, and assuming the required derivatives exist, define

$$\widehat{m}_i^{(1)} = m - \nabla f_i(O_i)^{\top} \Delta O_i, \tag{30}$$

$$\widehat{m}_i^{(2)} = \widehat{m}_i^{(1)} + \tfrac{1}{2} \nabla^2 f_i(O_i)[\Delta O_i, \Delta O_i], \tag{31}$$

$$\widehat{m}_i^{(3)} = \widehat{m}_i^{(2)} - \tfrac{1}{6} \nabla^3 f_i(O_i)[\Delta O_i, \Delta O_i, \Delta O_i]. \tag{32}$$

Each margin estimate is converted to a score using the same Bernoulli KL as in Section C.2. Thus, the approximation order changes the estimated downstream margin, not the evaluation of the KL itself. The first-order term uses the shared VJP. Under a candidate-wise directional-derivative implementation, the second- and third-order terms additionally require derivative contractions along each candidate's distinct perturbation direction. These contractions avoid materializing a full Hessian or third-order derivative tensor, but add computation and temporary storage. The conditional error bound in Section C.2 applies only to the first-order score.

Appendix D examines the scoring quality and computational cost of these extensions.

D Ablations and Sensitivity Analyses

D.1 Component Ablations

We evaluate the components of LazyRecall on all 500 problems in MATH-500 (Lightman et al., 2024), using DeepSeek-R1-Distill-Llama-8B. Each problem receives one generated response with a fixed random seed of 42, and accuracy is the percentage of correct final answers. The average generated response length is 3,455.4 tokens. All compressed configurations use an active GPU KV budget of $B = 1{,}024$ token-equivalent units. The full configuration sets $K = 16$ in the TopK retrieval operator for each layer–physical KV head pair. At each retrieval step, the total number of admitted KVs does not exceed 2% of the physical KV slots corresponding to budget B. All configurations use the same problems and generation settings.

We estimate recall-related CPU memory by subtracting the recall-disabled configuration's observed peak host-memory usage from that of the full configuration for each problem, then averaging the differences over the evaluation set. The estimate covers the CPU KV archive, FAISS indexes, position metadata, transfer buffers, and retrieval-process overhead.

Table D1 reports a progressive ablation. Starting from full LazyRecall, we first disable CPU recall, then additionally disable robust queue construction (temporal risk aggregation and neighborhood protection), and finally disable on-demand execution. When robust queue construction is disabled, base *Decision-Risk* scores still provide an eviction order. Base *Decision-Risk* scoring remains enabled in every compressed configuration. When on-demand execution is disabled, KVs are removed

Table D1: Progressive component ablations on MATH-500 with DeepSeek-R1-Distill-Llama-8B. Below the full LazyRecall configuration, each row additionally disables the component(s) named in its first column; earlier removals remain in effect. Dashes indicate inapplicable entries: component settings for FullCache and recall-specific CPU memory for configurations without recall. Boldface identifies the full LazyRecall configuration, not the best value in each column.

Configuration	Decision-Risk	Lazy exec.	Robust queue	Recall	Acc. (%) ↑	Recall CPU (GiB)	Peak GPU (GiB)	Throughput (tok/s)
FullCache	–	–	–	–	82.6	–	17.36	23.60
LazyRecall	✓	✓	✓	✓	**88.2**	**1.28**	**16.58**	**22.46**
Recall off	✓	✓	✓	×	86.0	–	16.48	22.69
Robust construction off	✓	✓	×	×	75.4	–	16.44	22.72
Lazy execution off	✓	×	×	×	70.6	–	16.47	22.77

in bulk to enforce the budget. FullCache is an uncompressed reference, so the component indicators do not apply to it.

Full LazyRecall achieves 88.2% accuracy, compared with 86.0% without recall. Further disabling robust queue construction reduces accuracy to 75.4%; subsequently disabling on-demand Lazy execution reduces it to 70.6%. Among these successive removals, disabling robust queue construction produces the largest accuracy drop, highlighting its contribution to generation quality in this setting. The full configuration has an estimated recall-related CPU-memory cost of 1.28 GiB. Its peak GPU allocation is 16.58 GiB, compared with 17.36 GiB for FullCache, while throughput is 22.46 versus 23.60 tokens/s.

D.2 Hyperparameter Sensitivity

We examine the audit window length W and recall intensity. The nominal window length W is the number of decoding steps for which an eviction queue is planned to be reused; an audit may occur earlier if too few valid candidates remain. The parameter K specifies how many candidates the TopK retrieval operator returns from each layer–physical KV head index. Retrieved candidates are not necessarily admitted: a separate cap limits the total number that may enter the active GPU cache.

We follow the MATH-500 evaluation setting in Appendix D.1, using DeepSeek-R1-Distill-Llama-8B and a fixed active GPU KV budget of $B = 1{,}024$ token-equivalent units. We report final-answer accuracy, recall-related CPU memory estimated as in Appendix D.1, peak allocated GPU memory, and generation throughput.

Audit window length. We vary W over 50, 150, and 250 decoding steps, while fixing the historical-query window, the retrieval parameter at $K = 16$, and the admission cap at 2% of the physical KV slots corresponding to B. As Table D2 shows, $W = 50$ achieves accuracy close to that of $W = 150$ but has substantially lower throughput. Increasing W to 250 slightly improves throughput while reducing accuracy. Memory usage changes little across these settings.

Table D2: Sensitivity to audit window length on MATH-500. Retrieval and admission settings are fixed.

W	Acc. (%) ↑	Recall CPU (GiB)	Peak GPU (GiB)	Throughput (tok/s)
50	88.0	1.34	16.65	20.36
150	88.2	1.28	16.58	22.46
250	84.6	1.28	16.57	22.73

Recall intensity. We fix $W = 150$ and jointly vary K and the admission cap. The settings $K = 8$, 16, and 40 are paired with caps of 1%, 2%, and 5%, respectively, of the physical KV slots corresponding to B. Table D3 shows that accuracy improves from the first to the second setting.

Expanding recall further does not improve accuracy and slightly reduces throughput; CPU and GPU memory usage change little.

Table D3: Sensitivity to recall intensity on MATH-500. Retrieval candidate count and admission cap vary jointly.

K	Admission cap	Acc. (%) ↑	Recall CPU (GiB)	Peak GPU (GiB)	Throughput (tok/s)
8	1%	87.0	1.28	16.50	22.49
16	2%	88.2	1.28	16.58	22.46
40	5%	88.2	1.29	16.69	22.32

Taken together, these results support $W = 150$, $K = 16$, and a 2% admission cap as the default configuration, balancing accuracy and throughput within the tested settings.

D.3 Decision-Risk Scoring Variants

We evaluate the two scoring extensions defined in Section C.4: enlarging the decision support and increasing the approximation order of the downstream margin change. We reuse the scoring diagnostics in Section 2.1, comparing each variant against measured full-vocabulary deletion KL using CV, Spearman's rank correlation, and the deletion-risk ratio. We also report the time required for one complete scoring pass.

Expanded Decision Supports. Table D4 reports the first-order variants. Here, $|S|$ is the number of tokens in the decision support; the current top-1 score uses $S = \{y\}$ and thus $|S| = 1$. For each top-p variant, we set the support threshold τ to the top-p value used by its generation configuration. All variants retain the first-order downstream approximation.

As the support expands, all three scoring diagnostics improve overall. At top-$p = 0.95$, however, the support contains 428 tokens on average and the deletion-risk ratio decreases only from 1.194 to 1.149, while scoring time rises from approximately 526 ms to 26,271 ms. The improvement in scoring quality is limited relative to the substantial increase in latency, which is impractical for periodic online risk audits.

Table D4: Effect of decision-support size on Decision-Risk scoring. All variants use a first-order approximation. CV is the coefficient of variation, ρ is Spearman's rank correlation, and RM is the deletion-risk ratio.

Configuration	$\lvert S\rvert$	CV ↓	ρ ↑	RM ↓	Time (ms) ↓
Top-1	1	0.054	0.304	1.194	525.59
Top-$p = 0.80$	53	0.046	0.342	1.163	4,207.81
Top-$p = 0.85$	84	0.045	0.346	1.160	6,842.03
Top-$p = 0.90$	141	0.045	0.354	1.155	11,546.56
Top-$p = 0.95$	428	0.042	0.370	1.149	26,271.14

Higher-Order Downstream Approximations. Table D5 keeps the single-token decision support $S = \{y\}$ fixed and changes only the approximation order of the post-removal margin. All orders use the same Bernoulli KL to obtain the final score. The second-order variant improves all three diagnostics over the first-order score. The third-order variant further reduces the deletion-risk ratio, but its CV and rank correlation are worse than those of the second-order variant. Scoring time rises from approximately 526 ms for first order to 51,501 ms and 331,930 ms for second and third order, respectively. Higher-order approximations therefore do not provide consistent scoring gains sufficient to offset their computational cost.

Taken together, the two extensions improve some scoring diagnostics, but the additional latency is prohibitive for periodic online audits. LazyRecall therefore uses the top-1, first-order Decision-Risk score described in the main text.

Table D5: Effect of downstream approximation order on top-1 Decision-Risk scoring.

Approximation	CV ↓	ρ ↑	RM ↓	Time (ms) ↓
First order	0.054	0.304	1.194	525.59
Second order	0.049	0.322	1.188	51,500.73
Third order	0.053	0.317	1.179	331,930.42

E Asynchronous Recall and GPU–CPU Coordination

E.1 Recall Coordination and State Consistency

The CPU archive and its FAISS indexes start empty and are updated as KVs are removed from the active GPU cache during generation. A separate CPU retrieval process maintains the indexes and performs maximum inner-product search (MIPS), while the GPU process handles decoding and risk audits. The key design principle is to keep CPU retrieval and host-to-device transfer off the current decoding critical path. As illustrated in Figure E1, GPU decoding and CPU retrieval proceed concurrently within each window. Candidates whose retrieval and transfer have completed are processed at a subsequent audit entry; unfinished retrievals never require GPU decoding to wait.

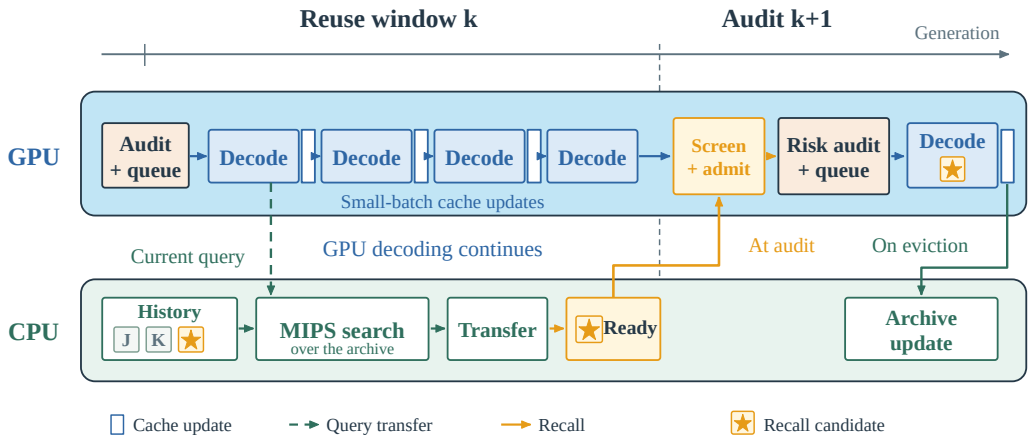


Figure E1: Asynchronous recall schedule. GPU decoding proceeds concurrently with CPU-side retrieval and host-to-device transfer. Queries are sent to the CPU within a window, while only candidates whose retrieval and transfer have completed are processed at a subsequent audit entry. Ready candidates are filtered and evaluated alongside resident KVs at a subsequent risk audit. Block widths do not represent measured durations.

Within-Window Queries and Indexed Retrieval. At fixed generation-step intervals, the system extracts the current token's post-RoPE queries, averages the query vectors sharing each physical KV head, and sends the results to the CPU. The retrieval process maintains a separate FAISS inner-product index for each layer–head pair, using `IndexIDMap2` around `IndexFlatIP` to associate keys with original positions. It searches FP32 keys for the top-K candidates and retrieves their states from separately stored BF16 K/V payloads. Here, K is the number of candidates returned per index; filtering and the admission cap may reduce the number evaluated alongside resident KVs. MIPS performs exact inner-product search over the current archive and serves only as a candidate-generation mechanism: retrieved states do not directly enter attention, but are supplied as candidates for a subsequent *Decision-Risk* audit.

Joining Ready Results At An Audit. Candidates are marked ready only after both CPU retrieval and transfer to the GPU have completed. At a subsequent risk audit, the system checks for the latest ready result, filters already-resident, duplicate, or invalid candidates, and applies a global admission cap. If no eligible candidates remain, the audit proceeds with resident KVs and updates the eviction

queue before decoding continues; otherwise, candidates from the GPU staging buffer undergo joint risk re-evaluation with resident KVs, and the eviction queue is then updated. Unfinished retrieval does not block the audit or decoding.

Recalled candidates from the GPU staging buffer participate in the audit forward pass and are scored jointly with resident KVs using the same shared VJP. Recall therefore requires no separate backward pass. MIPS similarity determines only which historical states are proposed for reconsideration; it does not determine which states remain resident. A retrieval hit does not guarantee subsequent retention: recalled and resident states compete under the same *Decision-Risk* criterion and the same budget. Retained candidates remain available for decoding until evicted again.

Identity and state consistency. The archive identifies each KV by its layer, physical head, and original token position, rather than its address after compaction. KVs removed during joint risk re-evaluation or subsequent on-demand removal are returned to the archive, including previously recalled candidates. Admission records prevent repeated use of old results, while buffer-completion events prevent premature payload overwrites. Appendix F describes the efficient transfer and cache-update paths.

E.2 Long-Run Stress Test and Latency Analysis

Long-Run Generation Stress Test. We use the Llama-3.1-8B-Instruct model and input from the inference-efficiency experiment in Section 4.3 to test continuous generation of 10,000, 50,000, and 100,000 output tokens. FullCache retains the complete KV history, while LazyRecall with recall disabled or enabled uses an active GPU KV budget of $B = 2{,}048$. To reach each target length, all configurations use greedy decoding without stopping at EOS. We set the retrieval parameter to $K = 16$. Each method–length combination is run eight times, and results are averaged.

Throughput is measured over the output tokens after step 2,048; time per token is calculated from the reported throughput. GPU memory is peak allocated memory. Recall-related CPU memory is estimated as in Appendix D.1.

Table E1: Long-run generation stress test. Results are averaged over eight runs for each method and output length.

Output tokens	Method	Throughput (tok/s) ↑	Time/token (ms) ↓	Peak GPU (GiB) ↓	Recall CPU (GiB) ↓
10,000	FullCache	24.654	40.561	16.269	–
	Recall off	23.970	41.719	15.376	–
	Recall on	23.695	42.203	15.392	2.907
50,000	FullCache	20.357	49.123	21.228	–
	Recall off	23.273	42.968	15.467	–
	Recall on	23.089	43.311	15.484	14.480
100,000	FullCache	15.232	65.651	27.427	–
	Recall off	23.432	42.677	15.470	–
	Recall on	23.111	43.269	15.485	29.037

All recall-enabled runs reach their target lengths while respecting the active GPU KV budget. Their peak GPU allocation and throughput remain nearly stable as output length increases. At 100,000 tokens, recall-enabled LazyRecall reduces peak GPU allocation by approximately 43.5% relative to FullCache while achieving higher throughput.

Latency Breakdown. Table E2 reports the mean time per triggered operation with recall enabled. GPU-to-CPU and CPU-to-GPU times are measured from sampled transfer batches. CPU search includes result packing, while the global risk audit includes the forward pass, shared VJP, scoring, and eviction-queue construction.

Transfer and global-audit times remain similar across output lengths. Although CPU search and packing become slower as the archive grows, their mean latency remains well below the decoding

Table E2: Latency breakdown with recall enabled. Each value is the mean time per triggered operation.

Output tokens	GPU-to-CPU transfer (ms/batch)	CPU search and packing (ms/call)	CPU-to-GPU transfer (ms/batch)	Global risk audit (ms/call)
10,000	0.159	71.062	0.142	372.605
50,000	0.161	466.850	0.148	358.303
100,000	0.161	768.324	0.152	357.273

time between successive risk audits. Much of this cost is hidden by asynchronous overlap with GPU decoding, and recall-enabled throughput remains nearly stable.

Together, the stress test and latency breakdown show that LazyRecall maintains stable operation under the tested long-run pressure, with Asynchronous Recall continuing to function throughout generation. This stability comes with a CPU-memory cost: at all three output lengths, recall-related CPU memory exceeds twice the reduction in peak GPU allocation relative to FullCache. When both GPU and CPU memory are constrained, we therefore recommend disabling recall; the ablation in Appendix D.1 shows that this configuration still maintains strong generation accuracy.

F CUDA Kernels and Efficient Implementation

LazyRecall introduces additional online computation for periodic *Decision-Risk* audits and dynamic cache updates during eviction and recall. We reduce these costs through fused CUDA kernels for risk computation, variable-length attention over irregular per-head caches, efficient cache-update and recall-merge operations, and bounded reusable workspaces. This appendix describes the main implementation optimizations used to reduce the computational and memory overhead of online KV-cache management.

Reordering Computation for Scalar Risk. Temporal aggregation ultimately uses attention outputs projected onto the gradient direction. We first compute $u_{r,i} = g_r^\top v_i$ and use

$$g_r^\top o_{s,r} = \sum_j \alpha_{s,r,j} u_{r,j}, \tag{33}$$

avoiding reconstruction of a high-dimensional output for every historical query before projection. For a physical head with n KVs of dimension d, R associated query heads, and T historical queries per query head, value-side computation decreases from $O(TRnd)$ to $O(Rnd + TRn)$. Query–key matching costs are unchanged.

Two CUDA kernels process the matrix-multiplication outputs. The first fuses scaling, causal masking, softmax, and scalar weighting; the second fuses deletion perturbations, GQA aggregation, temporal extrema, and risk evaluation. By the endpoint property in Appendix C, each candidate requires at most two nonlinear risk evaluations per temporal block. Threads access contiguous candidate positions and reduce values in registers and shared memory, avoiding the launches and memory traffic of separate tensor operations.

Variable-Length Caches and Fused Updates. Model-wide selection produces unequal cache lengths across physical heads. We store these as contiguous variable-length segments and invoke variable-length FlashAttention-2 using cumulative lengths while preserving GQA sharing (Dao, 2024; Ainslie et al., 2023). We design fused append and compaction kernels that jointly maintain K, V, and original positions. Compaction uses known deletion indices to determine the survivor layout, avoiding dynamic `nonzero` and repeated gathers. Recall merging writes surviving resident KVs and admitted candidates directly into the merged layout, avoiding a full concatenation intermediate. Surviving data still need to be moved, but index operations are not reconstructed separately for all three fields.

Bounding Audit and Transfer Workspaces. The vocabulary output head computes FP32 logits in blocks and analytically constructs the backward seed for the final hidden representation, avoiding

a temporary full FP32 output-weight matrix. Historical queries are also processed in workspace-sized blocks rather than shortening the history window for speed. Only audit steps build differentiable computation graphs; ordinary decoding uses the fast cache path. CPU archiving and recall use preallocated GPU staging buffers, registered shared memory, and a separate copy stream. Processes exchange buffer descriptors rather than serializing complete K/V payloads, and CUDA events control access and reuse. These optimizations reduce temporary allocations and blocking.

G Related Works

Prior KV-cache research addresses two complementary aspects of memory management. *Cache reduction* concerns what information should remain in the GPU cache and how limited capacity should be used, including representation compression, selection criteria, and budget allocation. *Online cache management* concerns how KV residency evolves during generation, with existing approaches broadly following eviction-based and retrieval-based designs. These two perspectives are complementary: the former determines how cache capacity is reduced or assigned, whereas the latter governs how the resident set changes over time and whether historical states remain accessible.

G.1 Cache Reduction

Cache-reduction methods operate along three main dimensions. *Representation compression* reduces the storage cost of retained KVs; *selection criteria* determine which KVs should receive limited cache capacity; and *budget allocation* distributes that capacity across model components.

Representation compression. KIVI and KVQuant exploit numerical properties of keys and values for low-bit representations (Liu et al., 2024c; Hooper et al., 2024). Palu reduces KV storage through low-rank projection, while MiniCache exploits cross-layer similarity (Chang et al., 2024; Liu et al., 2024a). These methods reduce the storage cost of retained states rather than primarily determining which states should be retained.

Selection criteria. Methods use different signals to identify KVs that should receive cache capacity. StreamingLLM exploits positional priors by retaining attention sinks and recent tokens, while SnapKV selects prompt KVs using observation-window attention and pooling (Xiao et al., 2024; Li et al., 2024). More recent methods incorporate output information into the selection criterion. ReST-KV models local output changes induced by removal, LAVa derives scores from an upper bound on attention-output error, and OBCache evaluates local attention-output perturbations (An et al., 2026; Shen et al., 2025; Gu et al., 2025). LaProx formulates attention-output approximation as a matrix-multiplication approximation problem and enables model-wide selection through layer-wise score normalization (Mai & Kim, 2026).

Budget allocation. Budget-allocation methods account for heterogeneous cache requirements across model components. PyramidKV varies cache allocation across layers, while HeadKV and Ada-KV allocate capacity across KV heads (Cai et al., 2024; Fu et al., 2025; Feng et al., 2026). Ada-SnapKV combines adaptive head-level allocation with SnapKV's window-based selection criterion (Feng et al., 2026; Li et al., 2024).

G.2 Online Cache Management

Online cache management focuses on how the resident KV set evolves as generation proceeds. Existing approaches broadly follow two directions: *eviction-based methods* dynamically determine when resident KVs leave the active cache, whereas *retrieval-based methods* maintain access to historical states outside the active GPU cache.

Eviction-based methods. Eviction-based approaches adapt the resident set to changing KV relevance during generation. RefreshKV updates the accessed historical subset as generation proceeds, while LazyEviction delays removal based on recurring attention patterns (Xu et al., 2025; Zhang et al., 2026). These methods reflect that KV importance is time-varying and that low current importance need not imply persistent irrelevance.

Retrieval-based methods. Retrieval-based approaches use host-side storage and retrieval to retain access to historical KVs beyond the active GPU cache. PQCache uses product quantization to identify and fetch relevant historical KVs (Zhang et al., 2025). RetrievalAttention uses CPU-side vector retrieval, MagicPIG performs CPU-side attention through sampling, and LouisKV refreshes retrieval results at semantic boundaries (Liu et al., 2026; Chen et al., 2025; Wu et al., 2026). These methods differ in how historical states are identified and incorporated into generation, but share the goal of extending accessible context beyond the GPU-resident KV set.