QUEEN: QUERY ENRICHMENT FOR KV CACHE EVICTION IN LONG REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Key-value (KV) cache eviction alleviates the memory bottleneck of long-reasoning inference by selectively retaining important KV pairs. Most eviction methods score each cached KV pair by the attention it receives from the queries of the most recent tokens. However, these scores can undervalue KV pairs that future decoding still needs, since the queries cover only recent context and, under causal masking, each cannot see later tokens in the window. To address these limitations, we propose **Que**ry **En**richment (**QUEEN**), a training-free method that refines the queries of the most recent tokens so that each reflects all of them (*intra-window refinement*), and reuses refined queries from earlier reasoning and the input prompt (*cross-context reuse*), with additional *batch-scalable reuse scheduling*, which reduces the peak GPU memory of the reused queries for higher-throughput serving. On three long-reasoning benchmarks, the proposed QUEEN consistently outperforms existing eviction methods. Especially on LiveCodeBench, it matches full-KV accuracy with $\mathbf{5.2}\times$ less KV memory and $\mathbf{4.1}\times$ higher throughput.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed in long-context settings, such as chain-of-thought reasoning (Wei et al., 2022; Guo et al., 2025), retrieval-augmented generation (Lewis et al., 2020), and multi-document reasoning (Yang et al., 2018), where the lengths of reasoning tokens can extend to tens of thousands (referred to as 'long reasoning'). At such lengths, the key-value (KV) cache becomes a major memory and bandwidth bottleneck at inference (Pope et al., 2023), motivating research on KV cache eviction, which reduces memory usage by selectively retaining important tokens under a fixed memory budget (Li et al., 2024; Zhang et al., 2023; Cai et al., 2025a).

Following prior work that views attention over the KV cache as retrieval (Liu et al., 2025a), we frame token selection for KV cache eviction as a retrieval problem; the KV cache serves as the corpus, and the goal is to retrieve the KV pairs likely to be needed during future decoding. Under this framing, eviction consists of three components: (1) the query used for scoring (*e.g.*, the observation-window queries (Li et al., 2024)), (2) the signals from the cached entries (*i.e.*, keys and values) used to assess relevance, and (3) the ranking strategy that determines which KV pairs are retained. Prior work has primarily focused on the latter two; (2) exploiting signals from the cached entries (Cai et al., 2025a;

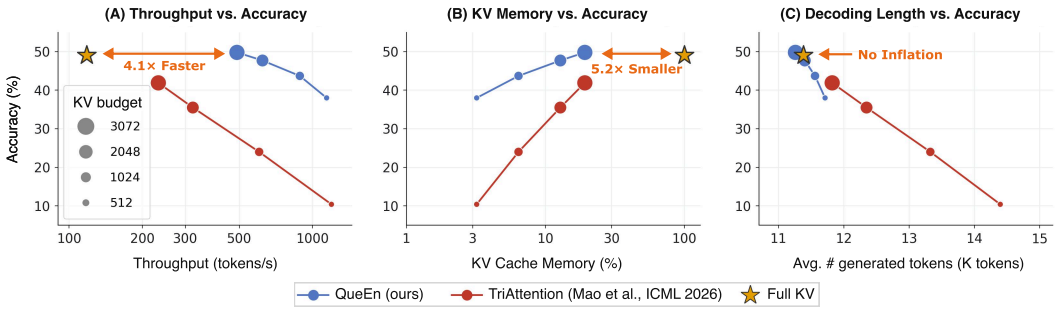

Figure 1: **Accuracy–efficiency trade-offs on LiveCodeBench** (Qwen3-8B). At the accuracy of Full KV, QUEEN **(A)** achieves $\mathbf{4.1}\times$ higher decoding throughput, **(B)** uses a $\mathbf{5.2}\times$ smaller KV cache, and **(C)** exhibits less decoding-length inflation (Tian et al., 2026) under compression.

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.

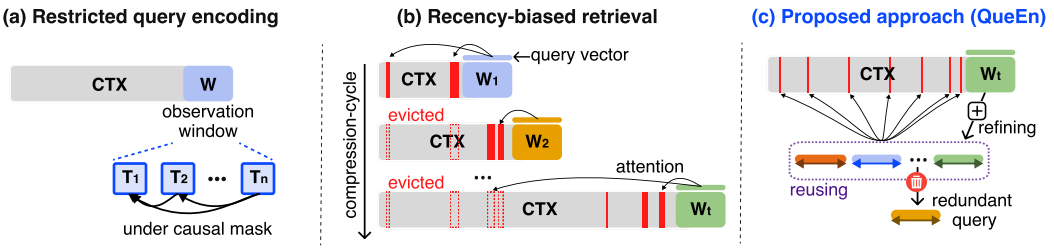

Figure 2: **Two limitations of observation-window scoring, and proposed QUEEN. (a)** The observation window W (tokens $T_1 \ldots T_n$) sits at the tail of the cached context (CTX). Under causal masking, the representation of each window token T_i is limited to its prefix context and cannot incorporate information revealed later in the window (Springer et al., 2025). **(b)** At each compression cycle, only the recent window provides query vectors ($W_1, \ldots, W_t$, each color marking the local context covered in that cycle), so the retrieval is biased toward the recent context; KV pairs from past contexts (red) needed later can be evicted (dashed). **(c)** QUEEN enriches the scoring query by *refining* and *reusing*: it refines the query vectors of the current window W_t to incorporate the completed window context, and reuses diverse past-window and question queries, discarding the redundant one by diversity selection.

Park et al., 2025; Guo et al., 2024) and (3) advancing ranking strategies (Feng et al., 2025; Cai et al., 2025b; Liu et al., 2025b). Yet the query is what defines the importance of each KV pair (Liu et al., 2025a). If it fails to assess the importance of a pair correctly, the pair can be evicted even though later decoding needs it. Here, we focus on revising or enriching the query with useful information for selecting KV pairs needed for future decoding.

To score KV pairs by query, recent eviction methods use the query vectors of the most recent tokens, *i.e.*, an *observation window* at the sequence tail (Li et al., 2024; Cai et al., 2025b; Liu et al., 2025b; Feng et al., 2025; Cai et al., 2025a). While simple and widely adopted, this observation-window query has two limitations. First, causal masking restricts the information that each query can encode. Under causal masking, the query vector of each token in the window is conditioned only on its prefix context and cannot incorporate information that appears later in the window (Springer et al., 2025) (Fig. 2-(a)), limiting retrieval effectiveness (BehnamGhader et al., 2024; Lee et al., 2025; Lin et al., 2026). Second, the observation-window query is semantically biased toward recent context. As the window query contains only the most recent tokens, the resulting scores favor recent KV pairs and may undervalue information from earlier contexts (Zhang et al., 2026). Since eviction is repeated throughout decoding, KV pairs from earlier contexts that are needed for subsequent decoding can be progressively evicted (Fig. 2-(b)), degrading performance (Guo et al., 2026; Zhang et al., 2026).

To address these limitations, we propose to enrich the scoring queries by refining them with the completed window context and reusing refined queries from earlier contexts, called **QUEEN** (***Que**ry **En**richment*), a training-free method for KV cache eviction (Fig. 2-(c)). QUEEN consists of two components. First, *intra-window refinement* encodes the window again after it is encoded (Springer et al., 2025), so that the query vector of each token in the window incorporates the full window context rather than only its prefix. Second, *cross-context reuse* extends query coverage beyond the recent window by retaining refined queries from past windows, selected layer-wise for diversity to cover distinct local contexts, together with the question query to preserve the global reasoning objective. To make query reuse practical, we further propose *batch-scalable reuse scheduling*, which reduces the peak GPU memory incurred by the reused queries for higher-throughput serving.

We empirically validate QUEEN across three long-reasoning benchmarks (MAA, 2025; Lightman et al., 2024; Jain et al., 2025), using three Qwen3 reasoning models (1.7B to 14B). QUEEN consistently outperforms existing methods across all benchmarks, with the gap widening as the budget tightens. In particular, on LiveCodeBench (Jain et al., 2025), QUEEN matches the uncompressed model's accuracy at $\mathbf{5.2}\times$ less KV memory and $\mathbf{4.1}\times$ higher throughput, while avoiding the decoding-length inflation induced by compression (Tian et al., 2026) (Fig. 1).

2 RELATED WORK

KV cache eviction. The mainstream approach to KV cache eviction, *observation-window scoring* (Li et al., 2024), scores cached tokens by the attention they receive from the queries in a short

window at the sequence tail. Subsequent work leverages key-value information in scoring, via key-similarity selection (Park et al., 2025) and value-based corrections (Guo et al., 2024), or advances the ranking strategy, via budget allocation across attention heads and layers (Feng et al., 2025; Cai et al., 2025b), and chunk-level selection (Liu et al., 2025b). Few works act on the query itself; LookaheadKV (Ahn et al., 2026) predicts future attention with learned lookahead modules. With reasoning models generating tens of thousands of tokens (Guo et al., 2025), the KV cache must be compressed repeatedly during decoding. Observation-window methods, such as R-KV (Cai et al., 2025a) and RPC (Song et al., 2025), evaluate token importance each cycle from the recent window alone, so their scoring stays biased toward recent context, and a token important in past cycles can be evicted even though it is re-attended later (Zhang et al., 2026). BeaconKV (Kim et al., 2026) mitigates this by maintaining beacon queries as representatives of anticipated future queries. TriAttention (Mao et al., 2026) replaces observation-window scoring altogether with trigonometric key scoring. QUEEN instead refines the queries so that they incorporate the completed window context, and reuses these refined queries to cover past reasoning contexts and the reasoning objective. Appendix G compares QUEEN with BeaconKV, the closest query-based baseline, in detail.

Retrieval over KV cache. Retrieval-based methods instead treat the KV cache as a searchable corpus, offloading it to host memory and fetching the query-relevant subset to the GPU at each decoding step, which avoids irreversible eviction but makes the limited host-to-GPU bandwidth a new bottleneck at every decoding step (Chen et al., 2024; Liu et al., 2025a; Sun et al., 2025). Such retrieval's quality hinges on the query: the better the query, the better the retrieval (Lee et al., 2024). QUEEN brings this query-centric retrieval view to the eviction setting, deciding what to retain with an enriched query.

Enhancing token representations in decoder-only LLMs. Causal masking limits each token representation to its prefix, which is known to limit embedding and retrieval quality (BehnamGhader et al., 2024; Lee et al., 2025). Prior work overcomes this limitation by removing the causal mask with additional training (BehnamGhader et al., 2024; Lee et al., 2025) or, without training, by repeating the input so that repeated tokens attend to the entire original sequence (Springer et al., 2025; Xu et al., 2024). QUEEN repurposes repetition for KV cache eviction, using it to refine the query vectors that score the cached keys rather than to obtain text embeddings or to enhance reasoning performance.

3 APPROACH

3.1 PRELIMINARIES

During long chain-of-thought reasoning with LLMs, the KV cache grows with the decoding length, making it a major memory bottleneck (Cai et al., 2025a). Under a fixed memory budget B, KV cache eviction (Liu et al., 2023) has been proposed to address this bottleneck by periodically discarding less important KV pairs. To determine which KV pairs to retain, most eviction methods assign importance scores to cached entries using a scoring query and evict the lower-scoring KV pairs. A common approach is *observation-window scoring*, which uses queries from a window W of size E at the sequence tail (Li et al., 2024; Cai et al., 2025b; Feng et al., 2025). Following this approach, for layer $\ell \in \{1, \dots, L\}$ and attention head $h \in \{1, \dots, H\}$, we define the importance scores $s_{\ell,h}$ of the cached keys as the maximum attention each key receives from the query vectors in W:

$$A_{\ell,h} = \mathrm{softmax}\big(Q_{\ell,h}\, K_{\ell,h}^{\top}/\sqrt{d}\big) \in \mathbb{R}^{E\times N}, \qquad s_{\ell,h} = \max_{i=1,\dots,E} A_{\ell,h}[i,:] \in \mathbb{R}^{N}, \tag{1}$$

where $A_{\ell,h}$ denotes the attention matrix, $Q_{\ell,h} \in \mathbb{R}^{E\times d}$ and $K_{\ell,h} \in \mathbb{R}^{N\times d}$ are the window query vectors and cached key vectors, d is the head dimension, and N is the number of cached keys. The eviction is governed by query vectors Q from the observation window W. As discussed in Sec. 1, however, these queries are limited by causal masking and biased toward recent context.

3.2 QUEEN: ENRICHING THE SCORING QUERIES

We address the limitations by proposing *intra-window refinement* (Sec. 3.2.1), which incorporates the completed window context, and *cross-context reuse* (Sec. 3.2.2), which extends query coverage beyond the recent context. We then describe how the enriched queries are used for eviction (Sec. 3.2.3).

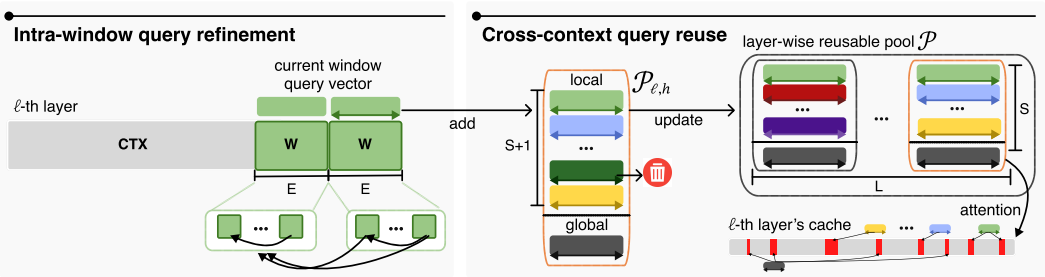

Figure 3: **Overview of QUEEN. Left** (*intra-window query refinement*): the E tokens of the window W are repeated at the tail of the cached context (CTX), so each refined query attends to the whole window. **Right** (*cross-context query reuse*): the pool $\mathcal{P}_{\ell,h}$ holds S past-window queries (local), dropping the most redundant one when they reach $S{+}1$, and the question query (global) that is always retained; the pooled queries score the cache, and the highest-scoring KV pairs (red) are retained.

3.2.1 INTRA-WINDOW QUERY REFINEMENT

The first limitation is caused by causal masking, which restricts query token representation to its prefix context, preventing it from incorporating information revealed later. To address this, we first propose to augment each query with the full window context, called *intra-window query refinement*. For each query q_ℓ^W, at each compression step, we feed the E tokens of the observation window W again as fixed inputs to the LLM immediately after the original window, and process them in a single additional forward pass (Fig. 3 left). Then, we extract the refined window query vectors $\tilde{q}_\ell^W \in \mathbb{R}^{H\times E\times d}$ at each layer ℓ, which hold one query vector per head over the E window tokens. Because the repeated window is positioned after the original one, each refined query can attend to the entire original window, enabling a *bidirectional-like* encoding without architectural changes or fine-tuning (Springer et al., 2025; Xu et al., 2024). We retain only these refined queries for scoring and discard the keys and values the refinement produces. Since the refinement processes a small window of E tokens in a single parallel forward pass, it adds only modest overhead (Sec. 4.5).

3.2.2 CROSS-CONTEXT QUERY REUSE

The second limitation is caused by scores relying only on queries from the recent window, which biases the retrieval toward recent context. As a result, KV pairs relevant to earlier contexts may be overlooked (Zhang et al., 2026). We address this by reusing earlier queries to preserve the KV pairs relevant to past local reasoning contexts and to the global reasoning objective, called *cross-context query reuse*. Specifically, it reuses refined *past-window queries* $\tilde{q}^W$ to cover earlier reasoning steps and a refined *question query* $\tilde{q}^Q$ to anchor the retrieval to the objective. At each layer ℓ and head h, we maintain these queries in a reusable *query pool* $\mathcal{P}_{\ell,h}$ for attention scoring (Fig. 3 right):

$$\mathcal{P}_{\ell,h} = \{\tilde{q}_{\ell,j}^W\}_{j=1}^{S} \cup \{\tilde{q}_\ell^Q\}, \tag{2}$$

where S is the maximum number of retained window queries and j indexes them. The pool is managed per layer, so all heads in layer ℓ share the same window queries.

(i) *Past-window queries.* To cover the local contexts of individual reasoning steps, we add the current refined window query $\tilde{q}_\ell^W$ to the pool $\mathcal{P}_{\ell,h}$ at each compression cycle. As this pool grows over cycles, we limit its size to S through *layer-wise diversity selection*. Because each layer represents the same window query differently, queries that are diverse at one layer may not be diverse at another, so we apply it independently at each layer. Specifically, we represent each pooled window query $\tilde{q}_{\ell,j}^W$ by its mean vector $\bar{x}_{\ell,j}$ over the E tokens of the window. To decide which window query to drop, we define the redundancy $r_\ell(j)$ as the total cosine similarity of $\bar{x}_{\ell,j}$ to the others, averaged over heads:

$$\bar{x}_{\ell,j} = \frac{1}{E}\sum_{t=1}^{E} \tilde{q}_{\ell,j}^W[:,t,:] \in \mathbb{R}^{H\times d}, \qquad r_\ell(j) = \frac{1}{H}\sum_{h=1}^{H} \sum_{j'\in[1:S+1]\setminus\{j\}} \cos\big(\bar{x}_{\ell,j,h},\, \bar{x}_{\ell,j',h}\big) \in \mathbb{R}. \tag{3}$$

When a new window query makes the pool reach $S{+}1$, we compute $r_\ell(j)$ over these $S{+}1$ window queries and drop the one with the largest $r_\ell(j)$ at each layer (Fig. 3 right).

(ii) *Question query.* The question, *i.e.*, the input prompt, states what the answer must satisfy and remains the single global objective of the reasoning trace, so the retrieval should be guided by it throughout decoding. However, the window queries alone cannot provide this global view, since each of them only reflects a local reasoning step. We therefore refine the query vectors of the question tokens at the first compression cycle and add them, $\tilde{q}_\ell^Q$, to the pool $\mathcal{P}_{\ell,h}$, retaining them throughout decoding without diversity selection. Since multi-step reasoning revisits KV pairs from many earlier reasoning steps (Zhang et al., 2026), we apply a *single-KV restriction*: each question-token query retains at most one KV pair, its highest-scoring one, so that the window queries can retain more KV pairs from the long reasoning trace.

Batch-scalable reuse scheduling. While reusing past queries provides access to earlier context, it also increases peak GPU memory for storing both the query pools and the query-by-key attention matrix. This overhead grows with batch size, limiting the larger batches needed for higher-throughput inference (Kwon et al., 2023). To address this limitation, we propose *batch-scalable reuse scheduling*, which keeps the query pools in host memory and fetches them to the GPU only when needed for scoring. The fetched queries then score the cache in tiles (Dao et al., 2022), with only the current tile's attention matrix kept on the GPU. This tiling is exact; since each key's score in Eq. 1 is a maximum over queries, taking the running maximum across tiles yields the same scores as processing the full pool at once while reducing peak memory during compression. Although query fetching and tiled scoring add modest overhead during compression, the reduction in peak memory allows larger batch sizes, thus improving end-to-end throughput (empirical validation in Sec. 4.5).

3.2.3 Evicting KV pairs with enriched queries

Once we enrich the scoring queries through *intra-window refinement* and *cross-context reuse*, they cover the completed window, the local contexts of past reasoning steps, and the global reasoning objective. For scoring, we compute the importance scores $s_{\ell,h}$ in Eq. 1 using the enriched queries in $\mathcal{P}_{\ell,h}$ instead of the original observation-window queries $Q_{\ell,h}$. Following prior work (Kim et al., 2025), we globally rank all KV pairs by these scores across layers, heads, and positions, retaining only the highest-scoring pairs under budget B.

We present the algorithm in Appendix A for the sake of space.

4 Experiments

4.1 Experimental Setup

Baselines and benchmarks. We compare against several baselines, including SnapKV (Li et al., 2024), R-KV (Cai et al., 2025a), BeaconKV (Kim et al., 2026), and TriAttention (Mao et al., 2026), of which only TriAttention requires offline calibration. We use their official implementations with the recommended hyperparameters (Appendix B). We evaluate all methods on three reasoning-capable Qwen3 models (Yang et al., 2025) at different scales: Qwen3-1.7B, Qwen3-8B, and Qwen3-14B. We use three long-reasoning benchmarks: AIME 2025 (MAA, 2025) and MATH500 (Lightman et al., 2024) for mathematical reasoning, and LiveCodeBench (Jain et al., 2025) for code generation.

Implementation details. QueEn has two hyperparameters, the refinement window length E and the pool size S; we set $E{=}32$ and $S{=}8$ in all experiments and analyze sensitivity to these choices in Sec. 4.5. The tile size of batch-scalable reuse scheduling is set to 64 (tile-width analysis in Appendix F.1). All methods compress the entire cache, input prompt included, to the same target budget B every 256 generated tokens, following Dong et al. (2026), except BeaconKV, which uses its original ratio-based schedule. Following prior work (Mao et al., 2026), all generations use the same temperature 0.6 and top-p 0.95, and each AIME problem is sampled 8 times with the average pass rate reported; MATH500 and LiveCodeBench are generated once per problem. The maximum generation length is 32K tokens for AIME and 16K for MATH500 and LiveCodeBench. All experiments are conducted on a single NVIDIA A6000 48 GB GPU. The full experimental setup is in Appendix B.

4.2 Results

Figure 4 reports accuracy across KV cache budgets on AIME 2025, MATH500, and LiveCodeBench for all three Qwen3 models. Our proposed QueEn consistently achieves the best compression

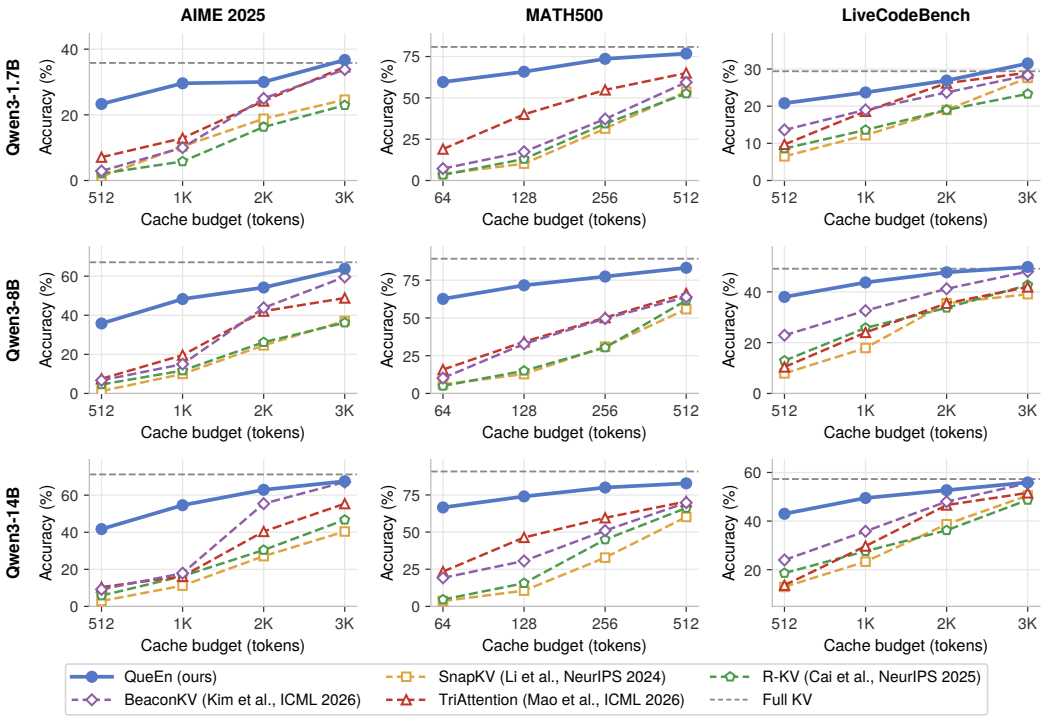

Figure 4: **QUEEN compared with SoTA eviction methods across various KV cache budgets.**

performance across all budgets on both math and coding benchmarks. Its advantage is particularly pronounced under tighter budgets, *e.g.*, reaching a 46.8 percentage-point gain over the strongest baseline, TriAttention, on MATH500 with Qwen3-8B at a 64-token budget. The same trend holds across additional backbones, including DeepSeek-R1 distilled models, Llama-8B, and Qwen-7B, where QUEEN again performs best at nearly every budget (full results in Appendix C.3). We attribute these gains to the two query enrichments. Intra-window refinement incorporates the completed window context for more informative KV selection, while cross-context reuse extends query coverage beyond the recent window, preserving KV pairs relevant to the local contexts of earlier reasoning steps and to the global reasoning objective.

4.3 ABLATION STUDIES

To assess each component, we ablate QUEEN on Qwen3-8B with a 128-token budget on MATH500 and a 1024-token budget on LiveCodeBench. As shown in Table 1, query reuse accounts for most of the gain, while query refinement alone provides only marginal gains but becomes far more effective when combined with reuse. The effects of the two enrichments are therefore synergistic rather than additive. We conjecture that reuse helps because the recent window alone covers neither the local contexts of earlier reasoning steps nor the global reasoning objective, while refinement strengthens reuse by improving all queries retained in the pool rather than only the current-window query.

Table 1: **Component ablation.**

Refine	Reuse	MATH500	LCB
✓	✓	**71.6**	**44.4**
✗	✓	62.4	34.8
✓	✗	11.0	22.9
✗	✗	10.4	21.1

Meanwhile, the single-KV restriction, which retains only each question query's highest-scoring KV pair, is ablated separately in Table 2: it improves MATH500 by +3.0 and LiveCodeBench by +3.9 points. We attribute this gain to the window queries preserving more of the reasoning contexts needed for subsequent decoding, using the budget the restriction frees (Appendix D relaxes the restriction to top-k KV pairs, lowering accuracy).

Table 2: **Single-KV restriction.**

Method	MATH500	LCB
QUEEN (full)	**71.6**	**44.4**
w/o restriction	68.6	40.5

4.4 Analysis of Query Enrichment Components

To understand the contribution of each query-enrichment component, we analyze them individually under compression. Specifically, we examine how query refinement improves KV selection, how reusing past-window and question queries preserves important KV pairs beyond the recent context, and how query diversity within the pool $\mathcal{P}$ strengthens reuse by retaining complementary queries.

Benefit of query refinement. To investigate the effect of query refinement, we evaluate it under *prefill* compression, where a single compression determines which KV pairs are retained without subsequent query reuse. Specifically, we measure how well the refined query preserves *oracle KV*, defined as the keys receiving the highest attention during full-cache decoding (Wang et al., 2025). We report oracle-KV recall as the fraction of oracle KV retained in the compressed cache at each budget (Appendix E.1). On Qasper (Dasigi et al., 2021), the refined query consistently achieves higher oracle-KV recall across all budgets than *vanilla window scoring*, which applies Eq. 1 with the original observation-window query (Fig. 5), indicating more effective KV selection. This improvement also leads to better downstream compression performance. On the 16 English tasks of LongBench (Bai et al., 2024), spanning long-context QA, summarization, few-shot learning, code, and synthetic tasks, query refinement alone achieves the highest average score among competing compression methods across the evaluated models under a 10% cache budget (Table 3; per-task results in Appendix C.1). Against vanilla window scoring, which uses the unrefined window query under the same setting, query refinement improves the average score in every model, showing that refinement helps even without query reuse.

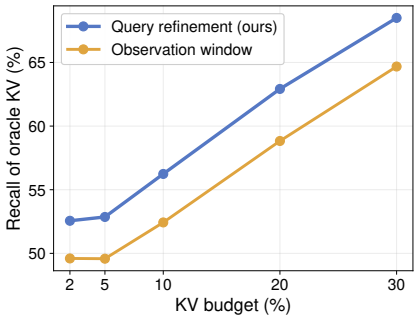

Figure 5: **Query refinement keeps more of the oracle KV at every budget** (Qasper, Qwen2.5-7B, n=200).

Table 3: **Effect of query refinement under prefill compression** (LongBench 16-task average, 10% KV budget).

Method	Llama-3.1-8B	Qwen2.5-7B	Qwen3-14B
No compression	45.1	48.6	50.2
SnapKV (Li et al., 2024)	43.3	42.2	48.4
ChunkKV (Liu et al., 2025b)	44.5	43.2	46.8
Vanilla window scoring	46.3	44.4	48.8
+ Query refinement (**Ours**)	**46.7**	**45.4**	**49.7**

Benefit of window-query reuse. To examine whether retrieval coverage beyond the recent context is needed, we analyze positions of *critical KV* over a reasoning trace of up to 32K tokens (AIME 2025, Qwen3-8B, budget 512). Following prior work that regards keys receiving high attention as critical (Zhang et al., 2023; Liu et al., 2023), we rank each key by the maximum attention it receives from decoding-step queries and define the top 512 keys as critical KV. Figure 6a shows the fraction of KV at each decoding position that is critical; critical KV is spread across positions rather than concentrated at recent positions. We then examine whether critical KV at all positions is retained after eviction (Fig. 6b). Observation-window scoring relies only on queries from the recent context, so it retains critical KV mostly at recent positions, whereas QueEn, reusing past-window queries that represent earlier reasoning states, retains critical KV across positions, and the same pattern appears in every layer and KV head (see Appendix E.2). Retaining critical KV beyond recent positions improves compression performance: QueEn reaches 35.8% accuracy while observation-window scoring drops to 1.2%, indicating the importance of the coverage that query reuse provides.

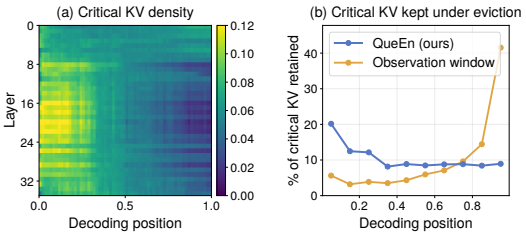

Figure 6: **Critical KV spans every decoding position, and observation-window scoring evicts the distant ones while QueEn retains them** (AIME 2025, Qwen3-8B, budget 512).

Benefit of question-query reuse. To investigate the benefit of question-query reuse, we examine whether it preserves KV pairs tied to the overall reasoning objective. As decoding proceeds, the recent window increasingly reflects local reasoning steps, so scoring guided by it alone can evict objective-relevant KV pairs still needed for the correct answer. Question-query reuse addresses this by anchoring the retrieval to the reasoning objective. To verify this, we analyze LiveCodeBench LeetCode tasks, where the function name specified in the problem must be reproduced exactly in the

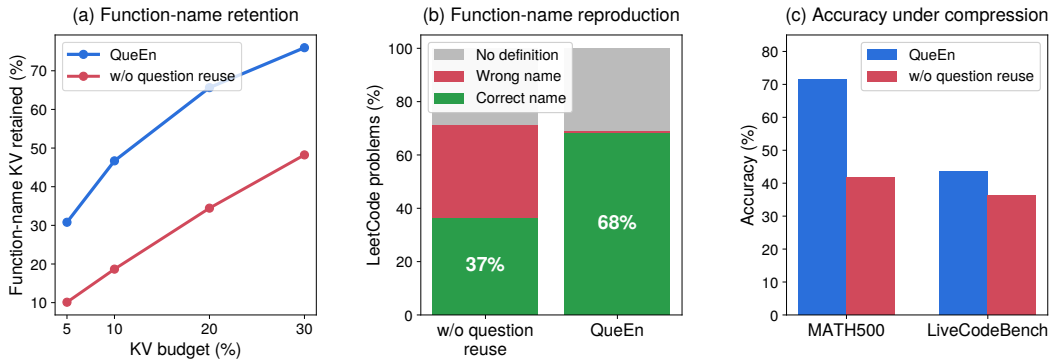

Figure 7: **Question-query reuse retains the function name the answer must reproduce, which window queries alone evict** (LiveCodeBench LeetCode, Qwen3-8B). **(a)** Without the question query the name is barely retained across budgets; **(b)** reuse reproduces the correct name where window queries alone hallucinate; **(c)** and it improves accuracy on MATH500 and LiveCodeBench.

generated answer. Its retention in the cache thus directly measures whether objective-relevant context is preserved. With question-query reuse, QUEEN retains more of these *function-name tokens* across all budgets (Fig. 7a), and correct function-name reproduction increases from 37% to 68% (Fig. 7b). Beyond this case, adding the question query improves accuracy under compression on MATH500 and LiveCodeBench (Fig. 7c), showing that retaining objective-relevant KV pairs leads to more accurate generation. Appendix E.3 presents a qualitative example of the hallucinated function name.

Benefit of window-query diversity. To examine whether diversity selection (Eq. 3) contributes to the accuracy gain, we replace it with a first-in-first-out (FIFO) pool of the same size (S=8), which retains only the most recent window queries. Figure 8 visualizes where the window queries pooled by diversity selection were extracted at the last compression cycle (Qwen3-8B, AIME 2025, B=2048, 32K-token generation limit), with darker cells indicating more queries from a given decoding position. Diversity selection draws window queries throughout the decoded sequence and retains different queries at each layer. Replacing diversity selection with FIFO reduces accuracy by 7.1 points (54.2→47.1), showing that query diversity broadens retrieval coverage and improves compression performance (same trend on the other benchmarks; Appendix D).

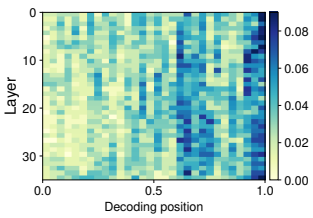

Figure 8: **Positions from which the pooled window queries were extracted**, at the last compression cycle.

4.5 EFFICIENCY AND HYPERPARAMETER SENSITIVITY

We evaluate the efficiency and robustness of QUEEN beyond task accuracy. We examine runtime efficiency, decoding-length inflation under compression, and sensitivity to key hyperparameters.

Runtime efficiency. We compare the maximum servable batch size and throughput of Full-KV, TriAttention, and QUEEN on Qwen3-8B under a decoding-heavy workload with a 256-token input, a 16,384-token decode, and budget B=3072 (Table 4). Compression increases the maximum batch size from 13 for Full-KV to 21 for TriAttention and 46 for QUEEN. Although TriAttention and QUEEN use the same cache budget, QUEEN supports a 2.2× larger batch because our proposed *batch-scalable reuse scheduling* reduces peak memory during compression. Without it (QUEEN †), the query pool $\mathcal{P}$ and the full query-by-key attention matrix must reside on the GPU, which lowers the maximum batch to 10 and the throughput to 178.4 tok/s. At its maximum batch size, QUEEN achieves 4.3× the throughput of Full-KV and 2.2× that of TriAttention (a comparison at the same batch size is given in Appendix F.2).

Table 4: **Efficiency comparison.** †: without batch-scalable reuse scheduling (Sec. 3.2.2).

Measure	Full-KV	TriAttn.	QUEEN †	QUEEN
Max batch	13	21	10	**46**
Throughput (tok/s)	122	234.9	178.4	**523.7**

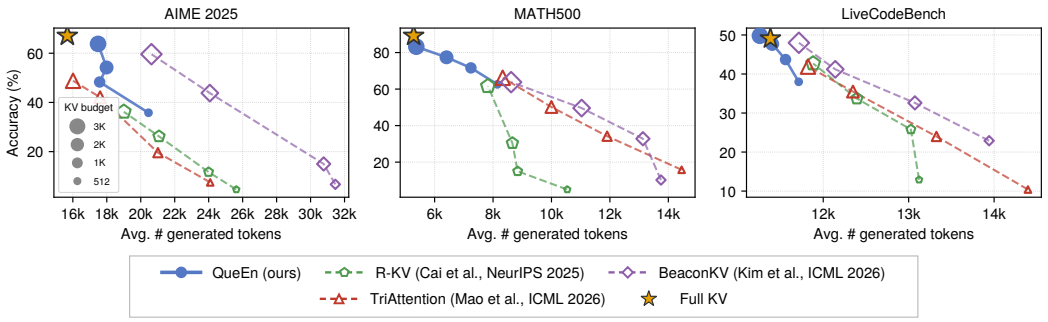

Figure 9: **Decoding-length inflation under compression.** Accuracy *vs*. average # of generated tokens across cache budgets on AIME 2025, MATH500, and LiveCodeBench (Qwen3-8B).

We then compare compression latency with TriAttention (Fig. 10), where QUEEN remains faster despite query refinement and reuse. Both enrichments add modest overhead because they are implemented as parallel matrix operations that utilize the GPU efficiently. Specifically, refinement processes the E tokens of the window in parallel, while query reuse adds rows to the same attention-scoring matrix multiplication in Eq. 1. Compared to QUEEN without query reuse, query fetching and tiled scoring increase per-compression latency by about 13%, while about a third of that latency is cache rebuilding, which enrichment does not affect (Appendix F.4). Because compression occurs only once every 256 decoding steps, the added overhead has little impact on overall throughput.

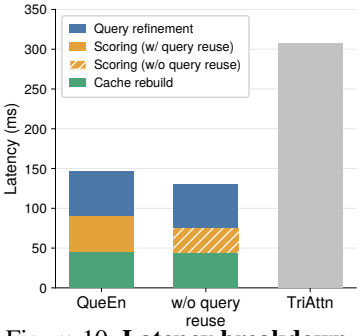

Figure 10: **Latency breakdown.**

Decoding-length inflation under compression. Aggressive compression can lengthen the reasoning chain as the model tends to overthink (Tian et al., 2026). To examine this phenomenon, we measure both the average number of generated tokens and accuracy across cache budgets on all three benchmarks (Fig. 9). QUEEN shows little decoding-length inflation while maintaining accuracy, whereas other methods generate substantially longer outputs and suffer larger accuracy drops under tight budgets. We conjecture that QUEEN helps preserve KV needed for reasoning, reducing the need to reconstruct lost intermediate information and thereby mitigating decoding-length inflation.

Hyperparameter sensitivity. We study the sensitivity to refinement window length E and pool size S, the number of window queries the pool holds (Table 5). On LongBench, accuracy is stable across E and peaks at E=32 (Qwen3-14B, 10% cache budget). On MATH500, accuracy plateaus beyond S=8 (Qwen3-8B, budget B=512). We therefore adopt these as defaults for all models and benchmarks, without any task-specific tuning.

Table 5: **Hyperparameter sensitivity.** Default settings are in **bold**.

Window length E	16	**32**	64	128
LongBench avg. (%)	49.00	**49.72**	49.47	49.29
Pool size S	2	4	**8**	16
MATH500 (%)	78.0	82.0	**83.2**	83.6

5 CONCLUSION

We viewed KV cache eviction as a retrieval problem over the cached KV and identified two limitations of the query that guides it: restricted query encoding under causal masking and a retrieval biased toward recent context. QUEEN addresses both through *query enrichment*, scoring the cache with refined and reused queries instead of the raw window queries. *Intra-window refinement* encodes the window again once it is complete so that each query reflects the whole window, and *cross-context reuse* maintains a diverse pool of past-window queries together with the question query, covering earlier reasoning steps and the global reasoning objective. To make query reuse practical, we propose *batch-scalable reuse scheduling*, which lets QUEEN serve larger batches. On three long-reasoning benchmarks, QUEEN consistently outperforms existing methods. In particular, on LiveCodeBench, it matches uncompressed accuracy at 5.2× less KV memory and 4.1× higher throughput. Our analysis attributes these gains to the enriched queries, which retain critical KV across all decoding positions and the KV pairs carrying the reasoning objective, both of which observation-window scoring evicts.

AI USE STATEMENT

In accordance with the ICLR 2027 policy, we declare that generative AI tools were not employed for any task subject to mandatory disclosure. Their use was confined to language editing, specifically the correction of linguistic errors and the refinement of clarity in text drafted by the authors. Each modification proposed by these tools was carefully examined by the authors to verify that the technical content and claims were preserved. The authors are solely responsible for the content of this manuscript, including any material prepared with AI assistance.

REPRODUCIBILITY STATEMENT

We have made several efforts to ensure the reproducibility of our results. Sec. 3.2 describes each component of QUEEN, and Appendix A provides the full procedure (Algorithm A.1) along with implementation notes. All hyperparameters (E=32, S=8, and a tile size of 64) are fixed across every model and benchmark (Sec. **??**). Appendix B details the datasets, the generation and evaluation protocol, the baseline configurations, and the hardware and software environment. All models and benchmarks we use are publicly available, and all baselines are run from their official implementations. We will release code publicly upon publication.

REFERENCES

Jinwoo Ahn, Ingyu Seong, Akhil Kedia, Junhan Kim, Hyemi Jang, Kangwook Lee, and Yongkweon Jeon. LookaheadKV: Fast and accurate KV cache eviction by glimpsing into the future without generation. In *International Conference on Learning Representations (ICLR)*, 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. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.172.

Parishad BehnamGhader, Vaibhav Adlakha, Marius Mosbach, Dzmitry Bahdanau, Nicolas Chapados, and Siva Reddy. LLM2Vec: Large language models are secretly powerful text encoders. In *Conference on Language Modeling (COLM)*, 2024.

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

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

Renze Chen, Zhuofeng Wang, Beiquan Cao, Tong Wu, Size Zheng, Xiuhong Li, Xuechao Wei, Shengen Yan, Meng Li, and Yun Liang. ArkVale: Efficient generative LLM inference with recallable key-value eviction. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

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

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Pradeep Dasigi, Kyle Lo, Iz Beltagy, Arman Cohan, Noah A. Smith, and Matt Gardner. A dataset of information-seeking questions and answers anchored in research papers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 4599–4610. Association for Computational Linguistics, 2021.

Zican Dong, Peiyu Liu, Junyi Li, Zhipeng Chen, Han Peng, Shuo Wang, and Wayne Xin Zhao. ForesightKV: Optimizing KV cache eviction for reasoning models by learning long-term contribution. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Zhenyuan Guo, Tong Chen, Wenlong Meng, Chen Gong, Xin Yu, Chengkun Wei, and Wenzhi Chen. Dynamic thinking-token selection for efficient reasoning in large reasoning models. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026.

Zhiyu Guo, Hidetaka Kamigaito, and Taro Watanabe. Attention score is not all you need for token importance indicator in KV cache reduction: Value also matters. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *NeurIPS Datasets and Benchmarks Track*, 2021.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations (ICLR)*, 2025.

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W. Lee, Sangdoo Yun, and Hyun Oh Song. KVzip: Query-agnostic KV cache compression with context reconstruction. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Janghyeon Kim, Minsoo Kim, Kyuhong Shim, and Jungwook Choi. BeaconKV: Key-value cache compression guided by beacon queries for efficient large reasoning model inference. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, volume 306 of *Proceedings of Machine Learning Research*. PMLR, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Symposium on Operating Systems Principles (SOSP)*, 2023.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. NV-Embed: Improved techniques for training LLMs as generalist embedding models. In *International Conference on Learning Representations (ICLR)*, 2025.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172. USENIX Association, 2024. arXiv:2406.19707.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri 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 (ICLR)*, 2024.

Ailiang Lin, Zhuoyun Li, Yusong Wang, Kotaro Funakoshi, and Manabu Okumura. Causal2Vec: Improving decoder-only LLMs as embedding models through a contextual token. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22769–22788. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.1042.

Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. RetrievalAttention: Accelerating long-context LLM inference via vector retrieval. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a.

Xiang Liu, Zhenheng Tang, Peijie Dong, Zeyu Li, Yue Liu, Bo Li, Xuming Hu, and Xiaowen Chu. ChunkKV: Semantic-preserving KV cache compression for efficient long-context LLM inference. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025b.

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. In *NeurIPS*, 2023. arXiv:2305.17118.

MAA. AIME 2025 problems (AIME 2025-I and 2025-II). `https://artofproblemsolving.com/wiki/index.php/2025_AIME_I_Problems`, `https://artofproblemsolving.com/wiki/index.php/2025_AIME_II_Problems`, 2025. Accessed: 2026-07-21.

Weian Mao, Xi Lin, Wei Huang, Yuxin Xie, Tianfu Fu, Bohan Zhuang, Song Han, and Yukang Chen. TriAttention: Efficient long reasoning with trigonometric KV compression. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026.

Junyoung Park, Dalton Jones, Matthew J. Morse, Raghavv Goel, Mingu Lee, and Chris Lott. KeyDiff: Key similarity-based KV cache eviction for long-context LLM inference in resource-constrained environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 5, 2023.

Jiwon Song, Dongwon Jo, Yulhwa Kim, and Jae-Joon Kim. Reasoning path compression: Compressing generation trajectories for efficient LLM reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Jacob Mitchell Springer, Suhas Kotha, Daniel Fried, Graham Neubig, and Aditi Raghunathan. Repetition improves language model embeddings. In *International Conference on Learning Representations (ICLR)*, 2025.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with Rotary Position Embedding. *Neurocomputing*, 568:127063, 2024.

Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. ShadowKV: KV cache in shadows for high-throughput long-context LLM inference. In *International Conference on Machine Learning (ICML)*, 2025.

Jiayi Tian, Seyedarmin Azizi, Yequan Zhao, Erfan Baghaei Potraghloo, Sean McPherson, Sharath Nittur Sridhar, Zhengyang Wang, Zheng Zhang, Massoud Pedram, and Souvik Kundu. SkipKV: Selective skipping of KV generation and storage for efficient inference with large reasoning models. In *Proceedings of the 9th Conference on Machine Learning and Systems (MLSys)*, 2026.

Yixuan Wang, Shiyu Ji, Yijun Liu, Yuzhuang Xu, Yang Xu, Qingfu Zhu, and Wanxiang Che. Lookahead Q-Cache: Achieving more consistent KV cache eviction via pseudo query. In *EMNLP*, 2025. arXiv:2505.20334.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Conference on Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations*, 2020.

Xiaohan Xu, Chongyang Tao, Tao Shen, Can Xu, Hongbo Xu, Guodong Long, Jian-Guang Lou, and Shuai Ma. Re-reading improves reasoning in large language models. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2024.

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

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

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. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.1683. arXiv:2506.15969.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

APPENDIX CONTENTS

A Algorithm and Implementation Details **14**
A.1 Algorithm . . . 14
A.2 Implementation notes . . . 15

B Experimental Details **15**
B.1 Datasets . . . 15
B.2 Generation and evaluation protocol . . . 16
B.3 Baseline configurations . . . 16
B.4 Experimental environment . . . 16

C Extended Results **16**
C.1 LongBench accuracy per task . . . 16
C.2 Decoding-time accuracy across budgets . . . 17
C.3 Results on additional models . . . 17

D Ablations of Design Choices **18**

E Details of the Query Enrichment Analysis **19**
E.1 Query refinement: oracle KV recall . . . 19
E.2 Query reuse: critical KV retention across layers and KV heads . . . 19
E.3 Query reuse: function-name case study . . . 20

F Efficiency Details **21**
F.1 Batch-scalable reuse scheduling: memory and latency of tiled scoring . . . 21
F.2 Throughput: isolating the effect of the larger batch . . . 22
F.3 Compression peak memory: extended analysis of the servable batch . . . 23
F.4 Per-compression latency: isolating the cost of query reuse . . . 23
F.5 Prefill-time efficiency . . . 23

G Comparison with BeaconKV **24**

H Limitations and Future Work **25**

A ALGORITHM AND IMPLEMENTATION DETAILS

A.1 ALGORITHM

Algorithm 1 gives the procedure of one decoding-time compression cycle at layer ℓ, mirroring the implementation.

Algorithm 1 QUEEN scoring at one compression cycle (per layer ℓ)

Require: Transformer f_{LM}; cache $\mathcal{C}$; observation window W ($|W|{=}E$); question tokens $\mathcal{Q}$; budget B
Require: persistent state: pool $\mathcal{P}_{\ell,h}$ for each head h (at most S window queries, the same across heads, and the question query)
if first compression cycle **then**
  input $\leftarrow [\,\mathcal{Q} \,\|\, W\,]$ ▷ question tokens prepended to the repeated window
else
  input $\leftarrow W$
end if
Forward the input through f_{LM} with $\mathcal{C}$ ▷ $\mathcal{C}$ is left unchanged
$\tilde{q}_\ell^W \leftarrow$ Refined queries of the current window W in the ℓ-th attention layer
if first compression cycle **then**
  $\tilde{q}_\ell^Q \leftarrow$ Refined queries of the question tokens $\mathcal{Q}$ in the ℓ-th attention layer
  add $\tilde{q}_\ell^Q$ to $\mathcal{P}_{\ell,h}$ ▷ kept for all later cycles; exempt from diversity selection
end if
append $\tilde{q}_\ell^W$ to $\mathcal{P}_{\ell,h}$
if $\mathcal{P}_{\ell,h}$ holds more than S window queries **then**
  remove the window query with the largest redundancy $r(j)$ ▷ diversity selection (Eq. 3)
end if
apply RoPE to each query in $\mathcal{P}_{\ell,h}$ at positions starting from the current sequence length
restrict each question query's row of $A_{\ell,h}$ to its highest-scoring key ▷ single-KV restriction
score the cached keys by Eq. 1, using the queries in $\mathcal{P}_{\ell,h}$
sort all keys by score across heads and layers; remove the lowest-scoring ones to meet budget B

A.2 IMPLEMENTATION NOTES

We note three implementation details. (i) The pool holds queries from earlier cycles as well as the current one, so we apply RoPE (Su et al., 2024) at the current sequence length, letting every query attend to the whole cache, including tokens formed after it. (ii) The redundancy in Eq. 3 is computed from the window query vectors alone, averaging each query over the E tokens of the window and then averaging the cosine similarity over heads. (iii) The question query is refined only at the first compression cycle, and the same vectors are reused at every later cycle.

B EXPERIMENTAL DETAILS

B.1 DATASETS

AIME 2025. All 30 problems of the 2025 American Invitational Mathematics Examination (MAA, 2025) (AIME 2025-I and 2025-II, 15 problems each). Every answer is an integer in $[0, 999]$, and we report exact-match accuracy on the final answer.

MATH500. The 500-problem subset of the MATH dataset (Hendrycks et al., 2021) curated by Lightman et al. (2024). We take the answer from the last `\boxed{}` and grade it by symbolic equivalence to the reference answer, up to LaTeX formatting.

LiveCodeBench. 279 problems from LiveCodeBench (Jain et al., 2025) (v5, collected 2024-08 through 2025-01), graded pass@1 by executing the official test cases.

LongBench. The 16 English tasks of LongBench (Bai et al., 2024), spanning single-document QA (NarrativeQA, Qasper, MultiFieldQA-en), multi-document QA (HotpotQA, 2WikiMultihopQA, MuSiQue), summarization (GovReport, QMSum, MultiNews), few-shot learning (TREC, TriviaQA, SamSum), synthetic tasks (PassageCount, PassageRetrieval-en), and code completion (LCC, RepoBench-P). We use these tasks for the prefill-time evaluation of query refinement in Sec. **??**.

B.2 Generation and evaluation protocol

All generations are sampled with temperature 0.6 and top-p 0.95. For AIME, each problem is sampled 8 times and we report the average pass@1; MATH500 is generated once per problem, following Mao et al. (2026), and LiveCodeBench likewise. We sample AIME 8 times because it has only 30 problems, whereas MATH500 (500 problems) and LiveCodeBench (279 problems) contain enough problems for a single generation each. The maximum generation length is 32K tokens for AIME and 16K for MATH500 and LiveCodeBench. During generation, the cache is compressed back to the target budget every 256 decoded tokens. The budget covers the entire cache: at each compression, KV pairs of both the input prompt and the generated tokens can be evicted. A generation that reaches the maximum length without producing a final answer is scored as incorrect. All numbers in this paper, including the uncompressed baseline, are produced by this single generation and scoring pipeline, so the comparison across methods is consistent.

B.3 Baseline configurations

All baselines are implemented following their official repositories with their originally recommended hyperparameters. For all methods except BeaconKV, we use a compression interval of 256 generated tokens, following Dong et al. (2026), so that they perform the same number of compression cycles over a generation. BeaconKV (Kim et al., 2026) is run through its official implementation with its original ratio-based schedule, which lets the cache grow $1/7$ beyond the target budget B and then evicts KV pairs to meet B. TriAttention (Mao et al., 2026) additionally requires offline calibration statistics for each model: we use the files released in its official repository, and for models it does not cover we run the calibration ourselves following the same protocol.

B.4 Experimental environment

All methods are implemented in the HuggingFace Transformers library (Wolf et al., 2020) and run in bfloat16 on a single NVIDIA A6000 (48 GB) GPU, with FlashAttention-2 (Dao, 2024) as the attention backend. Because eviction leaves each KV head with a different number of cached keys, decoding uses the variable-length FlashAttention kernel over the flattened cache.

C Extended Results

C.1 LongBench accuracy per task

Table 6 reports detailed LongBench results for all compression methods on Llama-3.1-8B, Qwen2.5-7B, and Qwen3-14B under a 10% cache budget. Query refinement attains the best 16-task average on every model, with the largest gains on the few-shot tasks.

Table 6: **LongBench per-task scores** (10% cache budget). Among compression methods, best results are in **bold** and second best are underlined.

Method	Single-Doc QA			Multi-Doc QA			Summ.			Few-shot			Synth.		Code		
	NrtQA	Qasp	MFQA	HotQA	2Wiki	MuSiQ	GovR	QMSum	MNews	TREC	TrivQA	SamS	PCnt	PRetr	LCC	RepoB	Avg
Llama-3.1-8B																	
No compression	30.5	47.3	55.0	58.6	51.6	33.0	35.0	25.0	27.0	29.0	84.8	39.3	10.2	100.0	51.4	44.2	45.1
SnapKV	30.9	37.6	52.7	56.8	45.7	27.8	25.8	23.9	19.5	46.5	78.5	39.5	12.5	**99.0**	46.4	50.1	43.3
ChunkKV	30.8	40.1	**56.2**	**57.8**	47.4	30.4	25.8	23.8	18.7	46.0	**83.5**	39.7	**14.5**	**99.0**	**47.8**	49.8	44.5
Query refinement (ours)	**31.9**	**45.3**	54.2	57.0	**49.8**	**32.1**	**29.0**	**24.7**	**22.9**	**64.0**	82.5	**42.0**	12.5	**99.0**	47.5	**52.2**	**46.7**
Qwen2.5-7B																	
No compression	29.3	46.1	50.3	55.6	43.3	28.1	33.8	24.3	25.5	67.5	88.1	41.9	11.5	100.0	68.3	64.2	48.6
SnapKV	**29.2**	35.8	44.9	49.3	32.1	22.6	27.1	23.5	17.2	45.5	**87.8**	41.6	7.0	98.8	49.2	**64.2**	42.2
ChunkKV	28.4	35.6	**50.9**	**52.7**	33.1	**24.1**	27.5	23.8	17.6	47.0	86.9	42.0	8.0	98.5	50.3	63.9	43.2
Query refinement (ours)	28.9	**42.3**	49.9	51.0	**36.8**	24.0	**29.0**	**24.0**	**20.8**	**65.0**	87.5	**43.0**	**9.0**	**99.5**	**51.9**	63.4	**45.4**
Qwen3-14B																	
No compression	30.2	44.2	52.4	62.1	56.1	32.9	32.6	24.3	24.7	70.0	89.6	42.1	9.2	99.4	68.2	65.7	50.2
SnapKV	**31.3**	39.7	50.4	**64.3**	49.5	**35.6**	26.3	24.2	17.7	67.2	89.9	41.7	6.8	97.5	**65.1**	66.5	48.4
ChunkKV	28.1	37.9	**50.7**	62.5	47.7	34.5	25.4	23.6	16.8	62.5	90.1	40.8	5.4	96.3	62.1	64.0	46.8
Query refinement (ours)	30.9	**41.9**	50.6	62.9	**54.4**	35.5	**27.9**	**24.4**	**20.5**	**74.5**	**90.3**	**42.8**	**9.2**	**98.4**	64.7	**66.6**	**49.7**

C.2 DECODING-TIME ACCURACY ACROSS BUDGETS

Table 7 reports the accuracy behind Figure 4 across cache budgets on AIME 2025, MATH500, and LiveCodeBench for Qwen3-1.7B, Qwen3-8B, and Qwen3-14B. QUEEN is the strongest compression method at every budget, benchmark, and model, and its advantage is particularly pronounced under tighter budgets. At the largest budget, QUEEN approaches the accuracy of the uncompressed model, and on LiveCodeBench it matches or exceeds that accuracy for two of the three models.

Table 7: **Decoding-time accuracy across cache budgets** behind Figure 4. AIME 2025 (30 problems) is the mean over 8 seeds; MATH500 (500 problems) and LiveCodeBench (v5, 279 problems, pass@1) are single generations. Among compression methods, best results are in **bold** and second best are underlined.

	Qwen3-1.7B				Qwen3-8B				Qwen3-14B			
AIME 2025												
Budget	512	1024	2048	3072	512	1024	2048	3072	512	1024	2048	3072
No compression			35.8				67.1				71.2	
SnapKV	1.2	10.4	18.8	24.6	1.2	10.0	24.6	37.1	2.9	11.2	27.1	40.4
R-KV	2.1	5.8	16.3	22.9	4.6	11.7	26.2	36.2	5.8	16.3	30.4	46.7
BeaconKV	2.9	10.0	25.0	33.8	6.7	15.0	43.8	59.6	9.2	17.9	55.4	67.1
TriAttention	7.1	12.9	24.2	34.6	7.5	19.6	42.1	48.8	10.4	16.2	40.4	55.4
QUEEN (ours)	**23.3**	**29.6**	**30.0**	**36.7**	**35.8**	**48.3**	**54.2**	**63.8**	**41.7**	**54.6**	**62.9**	**67.5**
MATH500												
Budget	64	128	256	512	64	128	256	512	64	128	256	512
No compression			80.8				89.2				90.8	
SnapKV	4.0	10.2	31.4	53.8	6.0	12.8	31.0	55.8	3.6	10.6	32.8	60.2
R-KV	3.4	13.0	34.4	52.6	5.0	15.0	30.4	61.6	4.6	15.6	45.0	66.2
BeaconKV	7.2	17.4	37.2	59.4	10.2	32.8	49.6	63.8	19.2	30.6	51.0	69.8
TriAttention	19.0	40.0	54.8	65.0	15.8	34.2	50.2	66.2	23.4	46.4	59.6	70.6
QUEEN (ours)	**59.6**	**65.8**	**73.6**	**76.8**	**62.6**	**71.6**	**77.4**	**83.2**	**66.6**	**74.0**	**80.0**	**82.8**
LiveCodeBench												
Budget	512	1024	2048	3072	512	1024	2048	3072	512	1024	2048	3072
No compression			29.4				49.1				57.3	
SnapKV	6.5	12.2	19.0	27.6	7.9	17.9	35.5	39.1	12.9	23.3	38.7	50.5
R-KV	8.6	13.6	19.0	23.3	12.9	25.8	33.7	42.7	18.6	27.6	36.2	48.7
BeaconKV	13.6	19.0	23.7	28.3	22.9	32.6	41.2	48.0	24.0	35.8	48.0	55.6
TriAttention	9.7	18.6	26.2	29.0	10.4	24.0	35.5	41.9	13.6	29.7	46.6	51.6
QUEEN (ours)	**20.8**	**23.7**	**26.9**	**31.5**	**38.0**	**44.4**	**47.7**	**49.8**	**43.0**	**49.5**	**52.7**	**55.9**

C.3 RESULTS ON ADDITIONAL MODELS

We validate QUEEN on two DeepSeek-R1-Distill models (Guo et al., 2025), Llama-8B and Qwen-7B, which are trained differently from the Qwen3 models evaluated in the main text. Llama-8B is further built on a different base model family (Llama). Table 8 reports the accuracy of both models across cache budgets on AIME 2025, MATH500, and LiveCodeBench against the same baselines as the main text. QUEEN achieves the best compression performance at nearly every budget, tying with TriAttention on AIME 2025 for Qwen-7B at B=3072 and trailing BeaconKV on MATH500 for Llama-8B at B=512, and its advantage is particularly pronounced under tighter budgets, reaching an 8.6 to 41.0 percentage-point gain over the strongest baseline at the smallest budget of each benchmark. The pattern matches the Qwen3 results, showing that the gains of QUEEN generalize across model families.

Table 8: **DeepSeek-R1-Distill models across cache budgets.** AIME 2025 (30 problems) is the mean over 8 seeds; MATH500 (500 problems) and LiveCodeBench (v5, 279 problems, pass@1) are single generations. Among compression methods, best results are in **bold** and second best are underlined.

	Llama-8B				Qwen-7B			
AIME 2025								
Budget	512	1024	2048	3072	512	1024	2048	3072
No compression	30.4				43.3			
SnapKV	5.8	14.2	21.2	26.2	7.5	20.4	29.2	35.0
R-KV	19.2	29.2	31.3	35.8	11.7	24.2	30.4	33.3
BeaconKV	18.3	30.0	31.2	34.2	16.7	22.9	32.5	35.7
TriAttention	20.0	27.9	32.9	31.2	17.1	27.1	33.8	**36.7**
QUEEN (ours)	**32.1**	**32.9**	**35.0**	**37.5**	**28.3**	**29.2**	**35.4**	**36.7**
MATH500								
Budget	64	128	256	512	64	128	256	512
No compression	82.6				85.4			
SnapKV	8.8	20.4	42.2	63.4	8.0	19.4	42.0	65.0
R-KV	11.8	38.4	66.0	77.8	8.4	24.2	53.8	69.4
BeaconKV	30.4	56.8	74.8	**81.2**	28.8	54.2	68.8	79.6
TriAttention	33.4	52.0	68.4	76.4	25.2	52.0	68.4	78.2
QUEEN (ours)	**71.8**	**72.6**	**76.8**	80.6	**69.8**	**76.4**	**82.6**	**83.0**
LiveCodeBench								
Budget	512	1024	2048	3072	512	1024	2048	3072
No compression	39.8				38.4			
SnapKV	11.5	17.9	30.5	31.9	8.2	15.1	23.7	28.7
R-KV	21.1	30.5	38.4	35.5	11.5	21.1	33.0	35.5
BeaconKV	22.9	31.5	37.3	37.6	16.1	27.2	29.0	34.4
TriAttention	12.2	25.4	36.2	35.8	10.0	24.4	35.1	35.1
QUEEN (ours)	**31.5**	**31.9**	**38.7**	**39.4**	**29.4**	**29.7**	**39.1**	**39.8**

D ABLATIONS OF DESIGN CHOICES

Layer-wise diversity selection. We extend the FIFO comparison of Sec. **??** to MATH500 and LiveCodeBench. Table 9 shows that diversity selection is worth $+7.1$ points on AIME 2025, $+1.2$ on MATH500, and $+1.7$ on LiveCodeBench. We conjecture that the gap is largest on AIME 2025 because its problems take far more decoding steps to solve; a FIFO pool, which holds only the most recent window queries, is then more limited in extending retrieval coverage.

Table 9: **Layer-wise diversity selection vs. FIFO pool** (Qwen3-8B, all other components fixed). AIME 2025 is the mean over 8 seeds; MATH500 and LiveCodeBench are single generations.

Benchmark	QUEEN (ours)	FIFO pool	Δ
AIME 2025 (B=2048)	54.2	47.1	$+7.1$
MATH500 (B=128)	71.6	70.4	$+1.2$
LiveCodeBench (B=1024)	44.4	42.7	$+1.7$

Single-KV restriction. In Sec. 4.3 we attributed the gain of the single-KV restriction to the budget it leaves to the window queries, which can then retain more KV pairs from earlier reasoning steps. We further ask whether one KV pair per question query is the right amount by letting each question query retain its top-k KV pairs instead (Qwen3-8B; MATH500 with B=128, LiveCodeBench with B=1024). On MATH500, accuracy decreases as k grows, from 71.6 at k=1 to 71.2 at k=2 and 70.4 at k=3, showing that the restriction is most effective when each question query keeps exactly one KV pair. On LiveCodeBench, accuracy is flat across k (44.4 at k=1 and k=2, 43.7 at k=3). Retaining more than one KV pair per question query does not improve accuracy on either benchmark (Table 10).

Table 10: **Relaxing the single-KV restriction to top-k** (Qwen3-8B). Each question query retains at most k KV pairs, its highest-scoring ones; k=1 is the setting used in QUEEN.

Benchmark	k=1 (ours)	k=2	k=3
MATH500 (B=128)	**71.6**	71.2	70.4
LiveCodeBench (B=1024)	**44.4**	**44.4**	43.7

E DETAILS OF THE QUERY ENRICHMENT ANALYSIS

E.1 QUERY REFINEMENT: ORACLE KV RECALL

Given a budget ratio ρ, each method keeps $k = \lfloor \rho N \rfloor$ of the N keys cached for the input context after prefill in each KV head. For layer ℓ and KV head h, let $\mathcal{O}_{\ell,h}$ be the set of k keys most attended during real decoding (the *oracle* "needed" set: top-k by the mean attention each key receives from the true decoding queries over the full KV), following the recall protocol of Wang et al. (2025), and $\mathcal{M}_{\ell,h}$ the set of k keys a method keeps (its top-k by importance). We then measure

$$\text{Recall@}k \;=\; \frac{1}{LH}\sum_{\ell=1}^{L}\sum_{h=1}^{H}\frac{\left|\,\mathcal{M}_{\ell,h}\,\cap\,\mathcal{O}_{\ell,h}\,\right|}{k}, \tag{4}$$

where L and H denote the numbers of layers and KV heads, respectively. $\text{Recall@}k = 1.0$ means that the method retains every key in the oracle set $\mathcal{O}_{\ell,h}$, so a higher recall means that fewer oracle keys are evicted.

E.2 QUERY REUSE: CRITICAL KV RETENTION ACROSS LAYERS AND KV HEADS

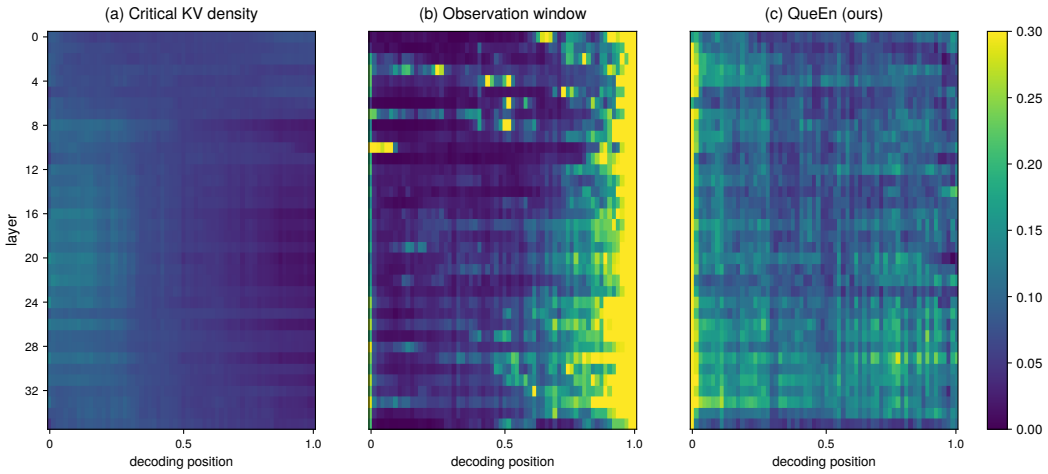

Figure 11: **Across all** 36 **layers, observation-window scoring retains critical KV mainly at the most recent positions, whereas QUEEN retains it across all positions** (AIME 2025, Qwen3-8B, B=2048, n=30). Rows correspond to layers and columns to 64 decoding-position bins, with values averaged over KV heads. All panels share the same color scale. **(a)** Density of critical KV. **(b)** Critical KV retained under observation-window scoring. **(c)** Critical KV retained under QUEEN.

To examine whether the retention gain in Figure 6b is consistent across layers, we repeat the measurement separately for each layer and KV head. The metric is unchanged: the fraction of critical KV at each decoding position that remains in the cache after eviction.

Figure 11 presents the results for all 36 layers of Qwen3-8B at a budget of B=2048, with decoding positions grouped into 64 bins and values averaged over KV heads. Panel (a) shows the density of critical KV. Panel (b) shows that observation-window scoring retains critical KV mainly at the most recent positions in every layer, while critical KV at earlier positions is rarely retained. Panel (c) shows

that QUEEN does not exhibit this recency concentration and retains critical KV across all positions and layers. Because this pattern holds in all 36 layers, the coverage gain in Figure 6b is consistent across layers rather than confined to a few of them.

Retention also varies across the KV heads within a layer. Figure 12 presents the same measurement per KV head for five sampled layers, where the pattern of Figure 11 also holds at the head level: observation-window scoring retains critical KV mainly at recent positions, whereas QUEEN retains it across positions.

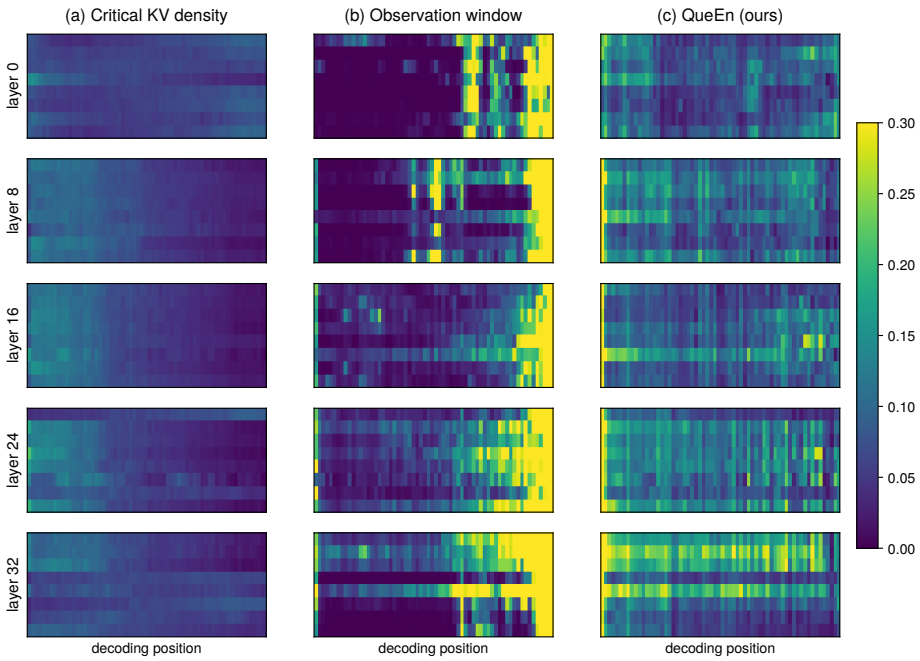

Figure 12: **The same pattern holds at the KV-head level** (AIME 2025, Qwen3-8B, B=2048; layers 0, 8, 16, 24, and 32). Rows are the 8 KV heads of each layer; panels are as in Figure 11.

Measurement details. Critical KV is defined as in Sec. **??**: for each layer and KV head, we rank the cached keys by the maximum attention they receive from the decoding queries during generation with the full KV cache, and take the top B, where B is the cache budget. Using the attention scores recorded during that generation, we then replay eviction cycle by cycle for both methods, so that they score identical keys at every cycle and differ only in the scoring query: the most recent window for observation-window scoring, and the diversity-selected pool for QUEEN. As in Figure 6, QUEEN here uses window-query reuse alone, without query refinement or question-query reuse, so that the comparison isolates the effect of reusing past-window queries.

E.3 QUERY REUSE: FUNCTION-NAME CASE STUDY

Table 11 shows an illustrative example, with both runs at the same budget (LiveCodeBench LeetCode, Qwen3-8B, B=3072). In LeetCode problems, the prompt includes *starter code*, a code template that fixes the class and function signature the solution must implement. Here, the starter code requires `class Solution` to implement `def maxLength(self, nums: List[int]) -> int`. Without question-query reuse, the scoring queries come only from recent windows, which reflect local reasoning steps rather than the global objective, so the KV pairs of the starter code can receive low scores and be evicted. The model then hallucinates the function name as `longestProductEquivalentSubarray`, while implementing the same algorithm. Since the official test cases call `maxLength`, which the generated code never defines, the submission fails every test even though the algorithm is correct. With question-query reuse, the objective stays in the retrieval, so the KV pairs of the starter code are retained, the model reproduces the correct function name `maxLength`, and the same algorithm passes all test cases. Similar hallucinations occur in other prob-

lems without question-query reuse (e.g., `countSubarrays` generated as `count_subarrays`, and `maxGoodNumber` as `maximum_binary_number`). This is reflected in the function-name reproduction rate, the fraction of problems whose generated code reproduces the function name specified in the problem: it rises from 37% to 68% with question-query reuse (Figure 7b).

Table 11: **Function-name hallucination without the question query** (LiveCodeBench LeetCode, Qwen3-8B, B=3072). The two solutions implement the same algorithm; only the signature differs.

Required (starter code)

```
class Solution:
    def maxLength(self, nums: List[int]) -> int:
```

Without question-query reuse (pass@1 = 0)

```
class Solution:
    def longestProductEquivalentSubarray(self, nums: List[int]) -> int:
        for i in range(n):
            current_gcd = math.gcd(current_gcd, nums[j])
            current_lcm = (current_lcm * nums[j]) // math.gcd(current_lcm, nums[j])
```

Full QUEEN (pass@1 = 1)

```
class Solution:
    def maxLength(self, nums: List[int]) -> int:
        for i in range(n):
            current_gcd = math.gcd(current_gcd, nums[j])
            current_lcm = (current_lcm * nums[j]) // math.gcd(current_lcm, nums[j])
```

F EFFICIENCY DETAILS

This section describes the measurement setup and reports the memory and latency analyses behind Table 4, together with prefill-time efficiency results. Unless otherwise noted, all measurements use Qwen3-8B with a 256-token input, a 16,384-token decode, a cache budget of B=3072, and a compression interval of 256 on a single A6000 GPU. We denote the batch size by b to distinguish it from the cache budget B. We measure the maximum batch size as the largest b for which the full 16,384-token decode completes, and confirm that running with b+1 exceeds the GPU memory.

F.1 BATCH-SCALABLE REUSE SCHEDULING: MEMORY AND LATENCY OF TILED SCORING

Computing the importance scores (Sec. 3.2.2) directly materializes an attention-score tensor of shape $[b, H, Q, N_k]$, where H is the number of attention heads, Q is the number of pooled query vectors, and N_k is the number of cached keys, so its peak memory grows linearly with Q. Batch-scalable reuse scheduling (Sec. 3.2) instead processes the queries in tiles of 64, in the style of FlashAttention (Dao et al., 2022), and updates a running maximum for each key, so the full tensor is never stored. The tiled computation is exact: the maximum over all pooled queries equals the maximum of the per-tile maxima, so the scores are identical to those of the materialized computation.

We profile the scoring step alone and report its peak memory as the number of pooled query vectors Q grows (Qwen3-8B: b=16, N_k=3072, head dimension d=128; Table 12). The values in the *Materialized* and *Tiled* columns are measured, not estimated from tensor shapes.

Table 12: **Peak scoring memory with and without query tiling** (Qwen3-8B, batch size 16, 3072 cached keys). *Materialized* forms the full attention matrix over all pooled queries, whereas *Tiled* processes them in tiles of 64.

# pooled queries	Materialized (GB)	Tiled (GB)	Reduction (GB)
512	2.14	0.33	1.8
1024	4.17	0.35	3.8
2048	8.23	0.38	7.8
4096	16.35	0.45	15.9
8192	32.59	0.58	32.0

The materialized computation stores the full attention-score tensor $[b, H, Q, N_k]$, so its memory doubles as Q doubles. The tiled computation holds at most one tile of 64 queries at a time, reducing

this tensor to $[b, H, 64, N_k]$, and keeps only the resulting maximum per cached key, $[b, H_{kv}, N_k]$, where H_{kv} is the number of KV heads, since the attention heads in a group share the same cached keys. The tiled memory still grows slightly, from 0.33 to 0.58 GB, because the queries of the layer being scored, $[b, H, Q, d]$, are held in GPU memory, while the pools of the other layers remain in pinned host memory. Most of the pool stays in host memory: at $Q{=}512$ it occupies 151 MB per sequence, a third of the 453 MB the cache holds at $B{=}3072$, but only the layer being scored, 4.2 MB per sequence, is on the GPU at any time, at the transfer cost reported in Appendix F.4. The memory saved by tiling, shown in the last column, grows with the number of pooled queries, from 1.8 GB at 512 pooled queries to 32 GB at 8192, so a large query pool can be used at large batch sizes without exceeding GPU memory.

Choice of tile width. We evaluate tile widths from 16 to 256. As shown in Table 13, width 64 attains the lowest latency for every number of pooled queries, with width 32 matching it at 512. As in FlashAttention (Dao et al., 2022), the tile width trades off the number of times the cached keys are streamed into on-chip memory against the shared memory each tile requires. Narrower tiles split the pooled queries into more tiles, and each tile streams the cached keys from GPU memory into on-chip memory again, so width 16 is up to $1.74\times$ slower than width 64. Beyond width 64, widening the tiles no longer reduces the latency: width 128 is slower by 1.10 to $1.18\times$, and width 256 exceeds the shared-memory capacity of the kernel. We therefore adopt a width of 64 in all experiments.

Table 13: **A tile width of 64 attains the lowest scoring latency** (Qwen3-8B). Latencies are relative to tile width 64 (**bold**); N/A marks widths the kernel cannot launch.

	Tile width				
# pooled queries	16	32	**64**	128	256
512	1.19×	1.00×	**1.00×**	1.10×	N/A
1024	1.35×	1.06×	**1.00×**	1.18×	N/A
2048	1.74×	1.06×	**1.00×**	1.17×	N/A

F.2 Throughput: isolating the effect of the larger batch

Table 4 reports each method at its own maximum batch size, so the throughput advantage of QueEn over TriAttention combines two effects: the larger batch QueEn can serve and the speed of its decoding loop. To isolate the second, we measure both methods at the same batch size, the maximum TriAttention supports ($b{=}21$), under the setup of Appendix F (Table 14).

At this batch size, QueEn attains 360.9 tok/s against 234.9 tok/s for TriAttention, a $1.54\times$ improvement, so the gain does not stem from the larger batch alone. The peak memory is also lower, 30.27 against 47.42 GB, and this margin allows QueEn to scale to $b{=}46$, where it attains 523.7 tok/s. Compression also lowers the cost of each decoding step: QueEn† serves a smaller batch than Full-KV (10 against 13) yet reaches 178.4 against 122 tok/s, because each of its decoding steps attends to at most $B{+}256$ cached keys, whereas Full-KV attends to the entire cache, which grows to $16{,}640$ tokens. All measurements use a HuggingFace Transformers stack rather than an optimized serving framework; we expect the same relative trends under systems such as vLLM (Kwon et al., 2023), since query enrichment is orthogonal to the attention kernel.

Table 14: **Throughput and peak memory by batch size** (Qwen3-8B, cache budget $B{=}3072$). The batch sizes $b{=}13$, $b{=}21$, and $b{=}46$ are the largest that Full-KV, TriAttention, and QueEn can serve, respectively. † denotes QueEn without batch-scalable reuse scheduling, whose largest batch is $b{=}10$.

	Full-KV ($b{=}13$)	TriAttn. ($b{=}21$)	QueEn† ($b{=}10$)	QueEn ($b{=}21$)	QueEn ($b{=}46$)
Throughput (tok/s)	122	234.9	178.4	**360.9**	523.7
Peak memory (GB)	48.74	47.42	45.12	**30.27**	46.83

F.3 COMPRESSION PEAK MEMORY: EXTENDED ANALYSIS OF THE SERVABLE BATCH

At a fixed budget, QUEEN and TriAttention store the same amount of KV cache, so the difference in maximum batch size in Table 4 arises from the temporary memory used *during* compression. The measured peak memory grows linearly with the batch size b (Table 15), and the KV cache accounts for 0.453 GB of the slope for both methods.

Table 15: **Peak memory during compression by batch size** (Qwen3-8B, B=3072). The slope is the memory added by each unit of batch size, of which 0.453 GB is the KV cache, the same for both methods.

Peak memory (GB)	b=4	b=8	b=12	b=16	Slope (GB per batch unit)
QUEEN	18.88	21.36	23.83	26.30	**0.618**
TriAttention	22.30	28.19	34.09	39.99	1.474

The rest of the slope is the temporary memory for compression, 0.165 GB for QUEEN against 1.021 GB for TriAttention, a $6.2\times$ difference that lets QUEEN serve a larger batch. The smaller temporary memory of QUEEN results from the tiled scoring of batch-scalable reuse scheduling (Appendix F.1). Because QUEEN never materializes the $[b, H, Q, N_k]$ attention-score tensor, scoring contributes only a single tile, $[b, H, 64, N_k]$, to the peak instead of the full score tensor.

F.4 PER-COMPRESSION LATENCY: ISOLATING THE COST OF QUERY REUSE

To measure the latency overhead of query reuse in one compression cycle, we compare two configurations that differ only in reuse (Qwen3-8B, batch size b=1, budget B=1024, 256-token question). Table 16 reports the measured latency breakdown behind the 13% overhead reported in Sec. 4.5. Without query reuse, scoring uses only the current window, E=32 query vectors. With reuse, it uses the S=8 pooled windows (8×32) together with the question query (256 tokens), for a total of 512. Even so, the end-to-end latency of one compression cycle increases only from 130.0 to 146.9 ms, or about 13%. The increase comes from fetching the offloaded queries ($+9.7$ ms) and from the tiled scoring of the pooled queries ($+7.8$ ms), while query refinement and cache rebuild are unchanged. Cache rebuild denotes applying the eviction: the surviving KV pairs are gathered into a new contiguous cache and the per-head offsets are recomputed, which costs the same in both configurations because they evict to the same budget. Timing the stages separately requires a device synchronization after each one, so the four stages sum slightly above the end-to-end total.

Table 16: **Per-compression latency with and without query reuse** (Qwen3-8B, batch size b=1, budget B=1024, 256-token question).

Latency (ms)	QUEEN	w/o query reuse	Δ
Query refinement	61.5	61.4	$+0.1$
Scoring	40.9	33.1	$+7.8$
Cache rebuild	49.8	49.6	$+0.2$
Query fetching	10.3	0.6	$+9.7$
End-to-end total	**146.9**	**130.0**	$+16.9$
Query vectors	512	32	$16\times$

F.5 PREFILL-TIME EFFICIENCY

Prefill-time compression applies only query refinement: after prefill, the last E=32 tokens are encoded again to obtain refined scoring queries, the cache is compressed once, and decoding proceeds on the compressed cache. Table 17 reports results in a long-input setting. Compressing the cache right after prefill increases the maximum batch size from 8 to 72 and the throughput by $7.0\times$ over full KV.

Table 17: **Prefill-time efficiency of query refinement** (Qwen3-8B, an 8,192-token prefill, a 256-token decode, 10% cache budget).

	Full-KV	**Query refinement (ours)**
Max batch	8	**72**
Throughput (tok/s)	110.3	**771.1**
Speedup		**7.0×**

G COMPARISON WITH BEACONKV

BeaconKV (Kim et al., 2026) is the closest prior work to QUEEN: both score the cache with queries from earlier positions of the sequence. BeaconKV builds 16 beacon queries from past queries so that they stand in for future global queries. QUEEN instead uses past queries to score the current cache and judge importance with respect to past reasoning contexts and the global reasoning objective. The purposes differ, but both retain past queries and score the cache with them. Our contribution is not scoring with historical queries itself, but the construction of *refined* scoring queries and an execution scheme that makes scoring with far more of them practical: under the setup of Appendix F, QUEEN scores with 512 queries where BeaconKV uses 32 per head. We describe the main differences between QUEEN and BeaconKV below.

Scaling the reused queries. BeaconKV keeps 16 beacon query vectors per attention head and adds 16 queries from the recent window, scoring the cache with 32 queries in total. This small query set keeps the scoring cost low, but makes increasing the number of reused queries costly. Each additional reused query occupies GPU memory, and at eviction the scoring step materializes the full attention matrix between all reused queries and the cache, so both memory and attention computation grow with the number of queries. When many sequences in a batch are compressed together, this growth raises the peak GPU memory and consequently limits throughput. QUEEN overcomes this limitation with batch-scalable reuse scheduling (Sec. 3.2). The pool is kept in host memory rather than on the GPU, fetched only when scoring takes place, and processed in tiles, so that only one tile's attention matrix resides on the GPU regardless of the pool size. This allows QUEEN to reuse many more queries than BeaconKV while still supporting efficient serving.

Query refinement. BeaconKV scores the cache with its 16 beacon queries and 16 recent-window queries. QUEEN instead refines the observation-window and question queries, aiming to overcome the restricted query encoding that limits these queries when they are used to score the cache. Our ablation shows that, under repeated compression during decoding, the effect of refinement is amplified when it is combined with reuse: refinement alone adds 0.6 and 1.8 points on MATH500 and LiveCodeBench, but 9.2 and 9.6 points when combined with reuse (Table 1). We attribute this difference to reuse: refinement then improves the representation of every query the pool retains, not only that of the current-window query.

Retaining the global reasoning objective. BeaconKV uses its beacon queries to look beyond the recent context and preserve the KV pairs that future queries revisit. QUEEN additionally places the question query in the reusable pool alongside the window queries, so that scoring reflects not only the importance of KV pairs to local reasoning steps but also the global reasoning objective throughout decoding. This mitigates the eviction of function-name tokens that the answer must reproduce (Sec. **??**, Fig. 7), and we conjecture that it also contributes to alleviating the decoding-length inflation under compression (Fig. 9).

Cost of query selection and maintenance. BeaconKV maintains 16 individual query vectors per head as beacons and selects geometrically distinct ones with continual FPS. As new queries are generated during decoding, FPS runs on every head every 16 decoding steps to update the beacons, regardless of when eviction occurs. Because the FPS schedule is independent of the compression schedule, FPS can run multiple times within a single compression cycle, incurring overhead separate from the cost of compression itself. On the released implementation, FPS accounts for 95% of

BeaconKV's overhead, making its per-step overhead the largest among the compared methods (Table 18).

In contrast, QUEEN uses the observation window as the unit of reuse. Diversity selection operates on the mean vectors of at most $S{+}1{=}9$ window queries (Eq. 3), so its cost depends on the number of windows the pool can hold ($S{=}8$) rather than on the number of window-query tokens ($9{\times}32$), and it runs only once per layer in each compression cycle.

This difference is reflected in serving throughput (Table 19). BeaconKV, whose scoring involves only 32 queries per head, supports a larger maximum batch than QUEEN (56 vs. 46). However, owing to the latency of query selection described above, its throughput remains lower than that of QUEEN despite the larger batch (367.8 vs. 523.7 tok/s), and the gap widens at the same batch size of 46 (347.4 vs. 523.7 tok/s).

Table 18: **Continual FPS dominates BeaconKV's compression overhead** (Qwen3-8B, $B{=}1024$, $b{=}1$, 4,096 decoding steps, excluding the decoding forward pass; BeaconKV in its released configuration). FPS runs 9,864 times across the 36 layers while eviction runs 21 times, so 1.94 of BeaconKV's 2.06 ms is continual FPS.

Method	Overhead per decoding step (ms)
TriAttention	1.26
BeaconKV	2.06
QUEEN	**0.82**

Table 19: **Maximum batch and throughput against BeaconKV** (Qwen3-8B, 256-token input, 16,384-token decode, $B{=}3072$). Each maximum batch is verified as in Appendix F; BeaconKV runs on its original schedule.

	Full-KV	TriAttn.	BeaconKV	QUEEN
Max batch b	13	21	**56**	46
Throughput at max batch (tok/s)	122	234.9	367.8	**523.7**
Throughput at $b{=}46$ (tok/s)	–	–	347.4	**523.7**

H LIMITATIONS AND FUTURE WORK

While QUEEN improves both accuracy and efficiency, two aspects remain open. First, query refinement requires an additional forward over the window, which adds to the cost of every compression cycle. A lightweight module trained to predict the refined queries from the window representations would remove this forward and further improve efficiency. Second, query reuse is confined to decoding time: at prefill, the queries of the input tokens are discarded rather than pooled. Reusing them, or predicting the queries of future decoding steps, is a natural direction for future work.