EPIKV: EPIPHANY-AWARE KV CACHE EVICTION WITHOUT THE ATTENTION MATRIX

Anonymous authors
Paper under double-blind review

ABSTRACT

Reasoning models can generate chains of thought tens of thousands of tokens long, making the key–value (KV) cache that holds them a major bottleneck for inference throughput. Existing eviction policies for long reasoning traces typically rank cached tokens using *attention weights*, requiring access to the attention matrix and making them incompatible with fast inference kernels. In this work we study the limits of such policies under tight cache budgets. Surprisingly, we find that under the strongest of them the generations that finish are wrong about as often as without eviction; most of the accuracy loss comes from generations that enter loops and run until the length limit, and retaining more tokens according to a fixed importance score exacerbates this behavior. What stops the looping is keeping the tokens the model's recent queries point to, and the forward pass the model already runs reveals them without the attention matrix. Motivated by this observation, we introduce epiphany-aware KV cache eviction (EPIKV), which combines hidden-state shifts with the model's recent query–key relevance to rank cached tokens *without materializing the attention matrix*. On multiple benchmarks, EPIKV matches or outperforms the strongest attention-based eviction baselines while running directly in vLLM with unmodified attention kernels.

1 INTRODUCTION

Reasoning models such as DeepSeek-R1 (Guo et al., 2025) solve hard problems by generating long chains of thought; for example, one math problem can take tens of thousands of tokens of reasoning before the answer. The key–value (KV) cache grows linearly with this length, which can dominate device memory across concurrent requests (Hooper et al., 2024) and limits the number of requests a server can hold (Kwon et al., 2023). KV cache eviction aims to address this by keeping only a budget of K tokens, but raises the question every eviction method must answer: *which tokens matter?*

Eviction methods built for reasoning traces often answer this via attention weights (Ramachandran et al., 2026; Hu et al., 2025; Su et al., 2026). Attention weight is a noisy proxy for importance (Jain & Wallace, 2019): attention sinks absorb weight whatever their content (Xiao et al., 2024), and the cumulative attention that H2O (Zhang et al., 2023) ranks by tracks counterfactual importance more

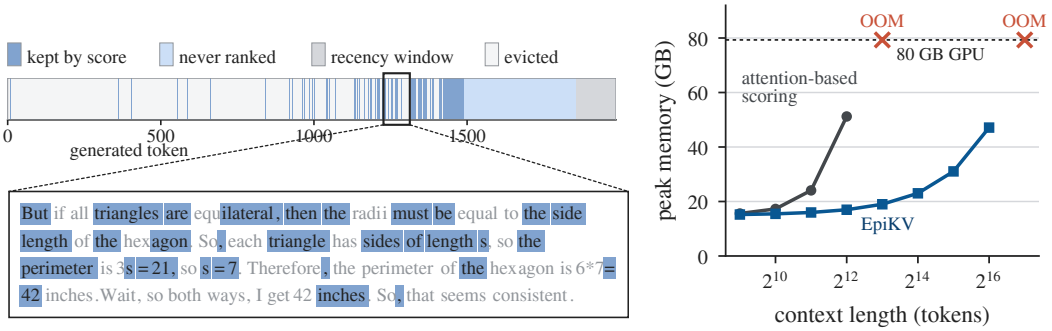

Figure 1: Left: the epiphany score keeps the concluded steps and the insight of a trace and evicts the connective prose between them. One MATH-500 trace under EPIKV^{-Q} at a 1024-token cache: the fate of every generated token, and the words of one stretch; the trace ends with the correct answer (Appendix C.1). **Right: reading the attention weights exhausts an 80 GB GPU at 8,192 tokens of prefill; FlashAttention, and so EPIKV, fits 65,536.** Peak memory of one forward pass (§3.3).

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.

weakly than the hidden-state signals we use (§2.2). It is also expensive to read. The weights exist only if the attention matrix is materialized, which efficient approaches such as FlashAttention are designed to avoid (Dao et al., 2022; Dao, 2024). Reading them forces the eager kernel, whose memory grows with the number of queries times the cached tokens: one pass exhausts an 80 GB GPU at a context shorter than a single reasoning trace (Figure 1, right), and a server that batches requests multiplies the queries. No paged serving engine exposes them at all. Without a custom kernel, the policies built for reasoning models thus may not be able to run in the engines that serve them. Attention-free scores exist, from the norm of the cached key (Devoto et al., 2024) to block statistics inside a paged cache (Chitty-Venkata et al., 2025), but they are fixed when a token is cached and blind to what the model is still using; the one we test, a fixed-budget form of LagKV (Liang et al., 2025), trails at tight budgets (§3.1).

To see what eviction costs in accuracy, we split every policy's errors on the same problems into generations that stopped with a wrong answer and generations that never stopped. Under RaaS (Hu et al., 2025), the strongest attention-based policy, and under our own policies, the generations that finish are wrong about as often as without eviction. Instead, we find that the extra errors are generations that loop until the length limit and never answer, under every policy and on all models we test (§3.2), a failure earlier work has observed under eviction (Hu et al., 2025; Chang et al., 2026). On one problem, a policy that ignores the model's recent queries lets it reach the right answer, restate it hundreds of times and write the digit 2 until the limit (Figure 4). Keeping more tokens does not help. Filling the cache with the next-best tokens by a score fixed when each token was generated makes the looping worse. By contrast, keeping the tokens the model's recent queries point to lets the generation stop. Which tokens matter therefore depends on which tokens the model is still using, and the model's own queries, read against the cached keys, identify them without the attention matrix.

This insight leads us to propose *epiphany-aware* KV cache eviction (EPIKV), which ranks tokens from the forward pass the model already runs. EPIKV scores each token once, when the token is generated, by how sharply it shifts the model's hidden states. The largest shifts mark an *epiphany* in the trace—turning points in the reasoning, such as a concluded step or a committed insight (Figure 1, left). Each time EPIKV evicts, it also compares the model's own recent queries against the cached keys, which tells it which cached tokens the model is still using. Neither step needs the attention matrix, a custom kernel, or training (Figure 2). We summarize our contributions below.

1. **Where eviction loses accuracy.** On both models and the same problems, we identify that the errors eviction adds at tight budgets are generations that never stop: most of them under every policy we test and nearly all under the strongest; filling the cache by a fixed score raises their number (§3.2, Table 3, Figure 4). The split costs nothing to compute, and we recommend that eviction papers report it beside accuracy (§5); the marked traces are released with the code.
2. **An eviction policy without the attention matrix.** On MATH-500, EPIKV matches the strongest attention-based policy at every budget on Llama-8B and exceeds every baseline at every budget on Qwen-7B; most of the problems its query term wins back are generations that now stop (§3.2, Table 3). On AIME 2024–2026 and GPQA-Diamond it is within paired noise of no eviction (Appendix D). It runs inside vLLM (Kwon et al., 2023) on stock kernels with the accuracy it has in our harness (§3.3, Table 4).
3. **The signal.** We find that hidden-state change in a mid band of layers marks a token's importance and change in an upper-mid band marks its absence, each tracking counterfactual importance more closely than cumulative attention, and that a causal rolling z-score removes the drift the residual stream carries over a long generation (§2.2). The labels are released with the code.
4. **Where reading the attention weights costs.** Crucially, EPIKV never reads the attention weights, so it runs on the kernels that serving engines use. We measure what reading them costs. It is little when one request decodes at a time, but at prefill and under batching the pass that returns them is slower and runs out of memory first (§3.3, Figure 1), and a policy that needs them cannot run in a paged serving engine without a custom kernel. Serving is where eviction matters most.

2 METHOD

2.1 PROBLEM SETUP

During autoregressive decoding the key–value (KV) cache grows by one entry per layer for every generated token. For a trace of n tokens over a model with L layers and H key–value heads of

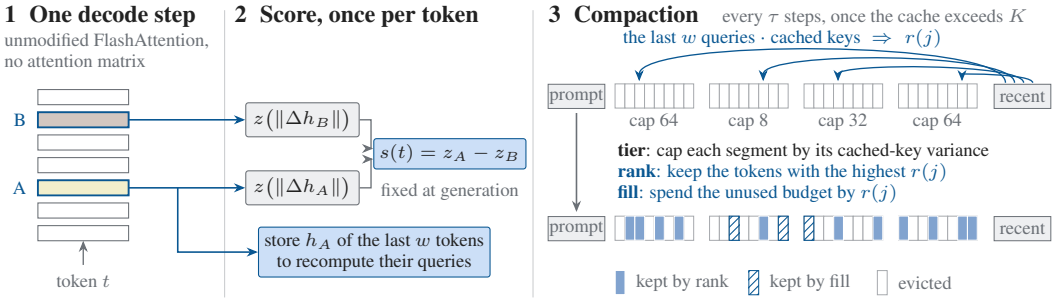

Figure 2: EPIKV reads what it needs from the forward pass; the attention matrix is never formed. (1) Decoding runs the unmodified FlashAttention kernel. (2) Each token gets a score $s(t)$, fixed at generation, from the hidden-state change at two layers. (3) Every τ steps over budget, segments are capped by tier and ranked by $r(j)$, the attention of the last w queries recomputed from Band-A states (EPIKV^{-Q} ranks by $s(t)$); positions are unchanged. Blue: what EPIKV adds.

dimension d_h, the cache holds $2LHd_hn$ scalars, and for traces of 10^4 to 10^5 tokens it dominates device memory. Decode-time eviction caps the cache at a budget of K tokens: at each compaction a policy scores the cached positions, keeps the K highest-scoring ones and discards the rest permanently. A retained token keeps its *logical* position: its position id and rotary offset (Su et al., 2024) continue from the original index, so eviction changes what the model attends to and not where the model believes it is. The alternative convention re-embeds the survivors at the compacted cache's physical slots; we call it the *rewound* convention. Several public implementations use it (Xiao et al., 2024), and §3.2 and Appendix A.2 show what it does to the results.

We call a policy *FlashAttention-compatible* (FA2-compatible) if it computes its score from two sources only: the cached keys and values, which already reside in high-bandwidth memory, and the per-layer hidden states that the standard `output_hidden_states` interface exposes. Such a policy never materializes the $n \times n$ attention matrix, so it runs inside a FlashAttention forward pass. A policy is *attention-based* if it needs that matrix. Reading the matrix disables FlashAttention's tiling and brings back $\mathcal{O}(n^2)$ peak memory for scoring (Figure 1, right).

2.2 THE EPIPHANY SCORE

Our signal is the per-token change in the residual stream. For layer l and decode position t, let $h_l(t)$ be the hidden state and define the L2 difference

$$g_l(t) = \|h_l(t) - h_l(t-1)\|_2. \tag{1}$$

A large $g_l(t)$ means that generating token t shifted the model's internal state at layer l. Such shifts mark consequential tokens, a committed step or an intermediate result, whereas fluent filler barely moves it; we call these turning points *epiphany* tokens and locate them below.

Measuring importance. To find which layers carry this signal we need a ground truth for a token's importance, and we obtain it by counterfactual occlusion (Appendix B.1). For each correctly answered trace we slide a 32-token window with stride 16 over the reasoning span, overwrite the window, regenerate the answer from the modified context, and label the window important if the answer changes. Every occlusion feeds the model the same context length, so the label measures content rather than position. The label is *recoverability under occlusion*: whether the model still reaches the answer with that span removed, which is the quantity eviction depends on at decode time. We label with the end-of-sequence filler and with three stop-free fillers, which agree with one another.

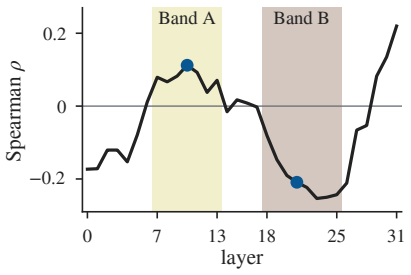

Figure 3: Hidden-state change marks importance in Band A and its absence in Band B. Spearman ρ with counterfactual importance per layer; Llama-8B, MATH-500. Dots mark the layers the score reads; all layers and AIME-2024 are in Table 11.

The two-band anatomy. Against these labels, two bands of layers behave consistently and in opposite directions on competition mathematics (Figure 3). In *Band A* (layers 7–13) the Spearman ρ between g_l and importance is positive at every layer, so a high g_l marks an important token. In *Band B* (layers 18–25)

it is negative at every layer, so a high g_l marks a dispensable token. The anatomy holds under both attention back-ends and under every measured filler, with smaller magnitudes under the stop-free ones (Appendix B.1); on AIME-2024 each band keeps its sign in the mean, with too few labeled traces for a per-layer claim (Appendix B.3). Each deployed band layer correlates with importance more strongly than cumulative attention, the signal H2O ranks by, on both configurations where attention weights exist (Table 12). We interpret the two bands, and how they move with the task, in §5.[1]

Positional drift in the residual stream. The magnitude of g_l is not stationary over a generation. Within a single trace, g_{10} drifts downward with position and g_{21} drifts upward, whatever the tokens contribute. A score read from absolute magnitude therefore tracks position as much as content, ranking early, droppable tokens above late, load-bearing ones. Every hidden-state signal we measure carries the drift, so it belongs to the residual stream and not to a scoring rule. A correlation with importance labels pooled over whole traces overstates how well such a signal ranks tokens *within* a trace.

Our score removes the drift by reading *local deviation* instead of magnitude. Each layer's difference is standardized against its own recent history by a causal rolling z-score,

$$z_l(t) = \frac{g_l(t) - \mu_l(t)}{\sigma_l(t) + \varepsilon}, \tag{2}$$

where $\mu_l(t)$ and $\sigma_l(t)$ are the mean and standard deviation of g_l over the trailing window of $W{=}64$ tokens ending at t, t included, and $\varepsilon{=}10^{-6}$; before the window fills, the statistics use the tokens generated so far, and the first decode token scores zero. The two bands combine into

$$s(t) = z_{10}(t) - z_{21}(t). \tag{3}$$

The window is causal, so a token's score depends only on positions up to t; tokens with high s are retained. The standardization adapts the lag-relative normalization of Liang et al. (2025) and the positional-bias correction of Gu et al. (2025) to hidden-state differences, and averaging over its window raises the correlation of every signal we deploy above its raw form (Table 14).

2.3 EPIKV

EPIKV holds a reasoning trace in a cache of K tokens by making three decisions at each compaction, all from tensors the forward pass already produces: how much of each 128-slot *segment* of the cache to keep, which tokens within a segment the model is still *reading*, and how to *spend* the budget the first decision leaves over. The second decision governs whether a generation stops (§3.2). The score $s(t)$ of Eq. (3) is computed when token t is generated and never revisited. Every $\tau{=}128$ decode steps, if the cache exceeds K, one compaction applies the three decisions and rewrites the cache. The prompt tokens and a window of the 128 most recent tokens are always kept.[2]

Segments and tiers. At each compaction the cache's decode span is cut into consecutive 128-slot segments. Each segment is classified as reasoning, execution or transition by the tertile of its mean key variance over the head dimension, the attention-free counterpart of ThinKV's segment sparsity, and is assigned a cap of 64, 32 or 8 retained tokens accordingly. Appendix C.2 compares two alternative segment signals, the entropy of that variance and ThinKV's attention sparsity itself.

The query term. Within a segment, tokens are ranked by how much the model's recent queries attend to them, read once per compaction and without the attention matrix. Let $t_1 < \cdots < t_w$ be the last $w{=}8$ generated tokens and ℓ the layer after the score's Band-A layer (layer 11 on Llama-8B, layer 21 on Qwen-7B). From the Band-A hidden state stored for each t_i, which is that layer's input, we recompute its query, $q_h(t_i) = \mathrm{RoPE}_{t_i}\big(W_h^q\,\mathrm{LN}(h_{\ell-1}(t_i))\big)$, and dot it with the cached keys of layer ℓ:

$$r(j) = \frac{1}{wH}\sum_{i=1}^{w}\sum_{h=1}^{H}\mathrm{softmax}_{j'\in\mathcal{C},\, j'\le t_i}\Big(\tfrac{q_h(t_i)\cdot k_h(j')}{\sqrt{d_h}}\Big)_j, \tag{4}$$

[1]The bands and the window are located once, on MATH-500, and applied unchanged to every other benchmark. Moving the band layers or the window changes accuracy within paired noise (Table 16).

[2]Methods differ in which structural tokens they preserve (attention sinks, a recency window, the prefill). Appendix A.3 tabulates each method's choice.

a causal softmax over the cached positions $\mathcal{C}$ for each query and head, averaged over both. The relevance r is z-scored over the cache and ranks the tokens within each segment. Computing it is one matrix product of w queries against the cached keys per head, on keys the compaction already gathers, once every 128 steps. Nothing is materialized inside the forward pass, and the kernel is untouched. The policy needs only the last w Band-A hidden states, two rolling buffers of W values and one score per cached token, a few hundred kilobytes for an 8B model.

The filled budget. The tier caps can bind before the budget does and leave part of K unused, up to about half of it on long traces. EPIKV spends that remainder on the evicted tokens with the highest r, so a compaction lands at K. Spending it by $s(t)$ instead costs accuracy and makes more generations loop (Appendix C.3), because a score frozen at generation time keeps tokens the model has stopped reading. The query term and the fill therefore ship together.

Table 1: Only our policies run on stock FlashAttention; every attention-based baseline needs the eager kernel or its own. *Eager* materializes the attention weights; *custom* needs a bespoke kernel, a serving-engine fork included; *stock* runs on unmodified FlashAttention-2. †A causal variant of Liang et al. (2025).

Method	Signal	Kernel
Attention-based baselines		
H2O	cumulative attention	✗ eager
ThinKV	attention sparsity per segment	✗ custom
RaaS	time each token was last attended	✗ eager
R-KV	attention − key redundancy	✗ eager
LongFlow	attention $\times \|v\|_1$	✗ custom
Ours: EPIKV *and its ablations*		
EPIKV	tiers, query term $r(j)$, fill	✓ stock
EPIKV$^{-\mathrm{Q}}$	tiers, ranked by $s(t)$	✓ stock
EPIKV$^{-\mathrm{QS}}$	global top-K of $s(t)$	✓ stock
raw score	global top-K of $\bar{g}_{10} - \bar{g}_{21}$	✓ stock
Ours: KV-vector signals		
KV-key	key variance	✓ stock
Lag-KV†	lag-normalized key and value	✓ stock

Compaction schedule. The cache is compacted at segment boundaries, every τ=128 decode steps, and only when it is over budget; τ=1 is the per-step form every baseline but R-KV uses. Between boundaries the cache can exceed K by up to $\tau-1$ entries, so the occupancy follows a sawtooth around the budget (Figure 5 shows it on one trace). The amortized form is the one a paged engine can run (§3.3). Appendix C.4 shows that compacting more often never improves accuracy and leaves the per-token cost unchanged.

2.4 ABLATIONS AND BASELINES

Table 1 lists every policy we evaluate. Three ablations of EPIKV remove its parts in turn; EPIKV$^{-\mathrm{Q}}$ appears in every main table and EPIKV$^{-\mathrm{QS}}$ in the appendix. EPIKV$^{-\mathrm{Q}}$ drops the query term and the fill: it ranks within segments by $s(t)$ itself and leaves the caps unspent. EPIKV$^{-\mathrm{QS}}$ drops the segments as well: it is a single global top-K over $s(t)$. The raw score is EPIKV$^{-\mathrm{QS}}$ without the detrending: a global top-K over $\bar{g}_{10} - \bar{g}_{21}$.

The KV-vector family scores tokens from quantities already in the cache and needs no hidden states. *KV-key* uses the rolling-mean variance of the key vectors across the head dimension; the value-vector and key-only lag variants are released with the code. *Lag-KV* brings the lag-relative normalization of Liang et al. (2025) into a fixed-budget decode policy. Each token's key and value vectors are normalized by the per-channel range of the *previous* chunk before the variance is taken, which removes domain-level shifts in magnitude, and every compaction ranks the whole cache against the budget K. LagKV instead normalizes against the *next* partition, keeps a fixed share of each partition per head and never revisits a compressed partition, so its cache grows with the sequence.

Every baseline runs in our harness at matched budgets; Appendix A.3 lists each policy's structural tokens and Appendix A.4 its adaptations.

3 RESULTS

EXPERIMENTAL SETUP

Models. DeepSeek-R1-Distill-Llama-8B (32 layers) (Guo et al., 2025) is the primary model, chosen for direct comparability with ThinKV and as an open-weight member of the reasoning-model class. DeepSeek-R1-Distill-Qwen-7B (28 layers; score layers 20 and 24, Appendix B.5) is the second

Table 2: EPIKV matches the best attention-based policy on Llama-8B and exceeds every baseline on Qwen-7B, at every budget. MATH-500 accuracy (%), n=495 per cell, greedy; every row runs the same problems on one platform. Bold: best eviction row in a column. †Above every attention-based row (exact McNemar $p < 0.05$; counts in Table 24). FA2: runs on FlashAttention. ⋆No eviction is budget-independent; most of its generations fit in 4,096 tokens (69% and 74%).

		Llama-8B				Qwen-7B			
Method	FA2	K=512	1024	2048	4096	K=512	1024	2048	4096
none⋆	✓	69.9				79.0			
EPIKV (ours)	✓	**66.5**	**70.3**	71.5	70.5	**66.1**	**75.2**†	78.0†	78.2†
EPIKV^{-Q} (no query term)	✓	57.8	63.2	68.9	71.3	63.0	70.7	**78.2**	**79.8**
Lag-KV	✓	52.7	60.4	66.9	69.7	34.5	53.5	70.9	77.6
RaaS	✗	64.8	69.9	**71.9**	71.7	65.3	69.9	73.1	72.5
H2O	✗	56.6	66.7	**71.9**	72.3	52.5	65.3	70.5	71.3
R-KV	✗	45.1	60.2	70.5	**72.5**	30.1	60.0	72.1	73.9
LongFlow	✗	44.0	62.4	70.1	72.1	33.1	58.4	71.7	72.9
ThinKV	✗	26.9	39.2	52.7	65.5	16.6	30.5	50.5	68.5

model in every accuracy table. Generation is greedy throughout, so repeated runs of one configuration agree exactly.

Datasets. MATH-500 (Hendrycks et al., 2021; Lightman et al., 2024) is the primary benchmark: competition mathematics with verifiable boxed answers and traces of several thousand tokens; we use the 495 problems that have a boxed reference answer. AIME 2024–2026 (Balunović et al., 2025), pooled over the three years, tests higher cache pressure with traces several times longer. GPQA-Diamond (Rein et al., 2024), graduate-level science as four-way multiple choice, tests a domain outside mathematics. GSM8K (Cobbe et al., 2021) serves only as a difficulty-regime probe for the layer anatomy (Appendix B.6), not as an accuracy benchmark.

Budgets and metrics. The cache budgets are $K \in \{512, 1024, 2048, 4096\}$ on MATH-500, $\{4096, 8192\}$ on AIME and $\{1024, 2048\}$ on GPQA-Diamond. Generation stops at the end-of-sequence token or at a cap of 8,192 tokens on MATH-500 and 16,384 on AIME and GPQA-Diamond. We report accuracy as exact match on the boxed answer, the number of generations that reach the cap, and wall-clock time and peak GPU memory per problem.

Attention back-end. Our policies run under FlashAttention-2 because they can; the attention-based policies run under the eager kernel because only it returns the weights they read. The two kernels compute the same attention, and every row scores the same problems. Every cell of the main accuracy tables ran on one platform under one library stack, the one on which our policies score lower (Appendix A.1); the hardware, the versions, and the ablations that ran elsewhere are listed there. The remaining results tables are in Appendix D.

3.1 ONE READ OF THE QUERY IS WORTH THE ATTENTION MATRIX

EPIKV matches the best attention-based policy at every budget on Llama-8B and exceeds every baseline at every budget on Qwen-7B (Table 2). Every row of the table runs the same problems, so every comparison is paired (Table 24). On Llama-8B EPIKV is within paired noise of RaaS and of the uncapped model at every budget. On Qwen-7B its lead over RaaS is significant from K=1024 upward, its lead over every other attention-based row is significant at every budget, and at the two largest budgets it is within paired noise of the uncapped model.

The query term recovers the tight budgets. Without it, EPIKV^{-Q} already leads Lag-KV at every budget on both models, and on Qwen-7B it leads the whole field from K=2048 upward and is level with RaaS at K=1024. On Llama-8B, however, RaaS leads it at the two tight budgets. EPIKV's wins over EPIKV^{-Q} concentrate there on Llama-8B and at K=1024 on Qwen-7B; elsewhere the two policies are within paired noise of each other. On Llama-8B the whole field converges at the largest budget, because most MATH-500 generations finish under it and eviction rarely fires. On Qwen-7B the attention-based rows stay below EPIKV even there.

EPIKV^{-Q}: cap hit at 8,192 tokens

```
So, 2220 is the smallest four-digit number. [...] So,
is 2220 the answer? Wait, but 2220 is 2220, but 2220
is 2220, which is 2220, which is 2220, which is 2220,
[... "which is 2220" 279 times ...] So, the number is
22222222222222 [... to the cap]
```

EPIKV: stops at 3,505 tokens

```
[...] 2220 is 2220, which is 2220/30 = 74, which is
integer. So, 2220 is the first four-digit number that
works. Therefore, 2220 is the smallest positive
integer multiple of 30 that can be written with only
the digits 0 and 2. [...] \boxed{2220}
```

Figure 4: The model reaches the answer and cannot stop. MATH-500 problem 10 (the smallest multiple of 30 written with only the digits 0 and 2), Llama-8B, K=1024. Without the query term the model restates the correct answer, then writes the digit 2 until the cap; with it, it checks the answer and stops (Appendix E.1).

Which attention-based policy sets the bar depends on the budget. At the two tight budgets RaaS is the strongest attention-based row on both models; at the loose budgets the attention-based rows converge, and on Llama-8B H2O ties or passes RaaS. RaaS keeps a token while the current queries attend to it, so it knows how relevant each cached token is to what the model is doing now, which a score frozen at generation time cannot know. RaaS pays for that signal with the attention matrix at every step. The hidden-state score gives up the query to avoid that cost, and EPIKV restores it with one read per compaction (§2.3), which recovers RaaS's margin at the tight budgets.

ThinKV is last at every budget on both models, and neither its refresh window nor its classifier explains its place on its own (Appendices A.5 and C.2). R-KV and LongFlow run on the same problems and platform as every other row. On Llama-8B both trail RaaS and H2O at every budget below the largest, on Qwen-7B both trail EPIKV^{-Q} at every budget, and both need the attention probabilities at every compaction (Table 1). To see where the accuracy lost at tight budgets goes, §3.2 splits the per-problem records by how each generation ended.

3.2 ACCURACY IS LOST TO LOOPS, NOT WRONG ANSWERS

At tight budgets eviction costs accuracy by breaking termination. We split every error into two kinds. A *cap hit* is a generation that reached the length cap without an answer, the signature of a model that has begun to repeat itself. A *terminated wrong answer* is a generation that stopped on its own and answered wrongly. VaSE traces such loops to the eviction of large-magnitude values (Chang et al., 2026); protecting those values changes neither our loops nor accuracy (Appendix E.3). Done for every policy on the same problems, the split measures how much of the lost accuracy the loops account for; Table 3 gives it at K=1024. RaaS, the hidden-state policies and the uncapped model answer almost the same number of terminated generations wrongly, and the whole of EPIKV^{-Q}'s excess error over the uncapped model is cap hits. ThinKV is the exception: it adds terminated wrong answers in large numbers as well as loops. Figure 4 shows one loop, in which the model reaches the right answer and cannot stop.

Table 3: Eviction loses accuracy by failing to stop, not by answering wrongly. MATH-500 errors at K=1024, n=495; a cap hit ran to the 8,192-token cap. Bottom: problems right under the first policy and wrong under the second; in parentheses, those where the wrong side hit the cap.

	Llama-8B			Qwen-7B		
	errors	cap hits	term. wrong	errors	cap hits	term. wrong
none	149	89	60	104	55	49
EPIKV	147	91	56	123	73	50
EPIKV^{-Q}	182	122	60	145	93	52
RaaS	149	87	62	149	99	50
H2O	165	97	68	172	124	48
ThinKV	301	158	143	344	191	153
EPIKV, not EPIKV^{-Q}	54 (40 looped)			38 (29 looped)		
EPIKV^{-Q}, not EPIKV	19 (17 looped)			16 (12 looped)		
RaaS, not none	42 (32 looped)			18 (15 looped)		
none, not RaaS	42 (31 looped)			63 (55 looped)		

The same accounting explains how an eviction policy can match the uncapped model at a tight budget. RaaS ties the uncapped model on Llama-8B at K=1024 with nearly the same count of terminated wrong answers, and most of the problems the two trade are generations that loop under one cache and stop under the other (Table 3). Which generations loop therefore depends on the cache, and a tie in accuracy can hide dozens of problems that change sides.

The query term of EPIKV acts through this channel and through no other. At K=1024 it lowers EPIKV^{-Q}'s cap hits on both models and leaves its terminated wrong answers where they were, and most of the problems it wins from EPIKV^{-Q} are generations that hit the cap under EPIKV^{-Q} and

now stop (Table 3). Its terminated wrong answers match RaaS's and the uncapped model's, and on the problems the uncapped model solves it loops on the same share on both models, whereas RaaS loops more often on Qwen-7B (Appendix E.4).

Doubling the generation cap leaves both policies' accuracy within paired noise, and most generations that reach the shorter cap reach the longer one too (Appendix E.2), so most cap hits are loops that more length would not finish. Nor is the remedy simply to keep more. Filling EPIKV^{-Q}'s cache to budget with the next-best tokens by the frozen score, everything else fixed, raises its cap-hit count and lowers its accuracy in most of the cells we ran, on both models; EPIKV spends the same budget by the query term instead (Appendix C.3). Accuracy follows the tokens the model is still reading when it decides to stop.

Under the rewound convention of §2.1 the failure is total: H2O loops on every problem at the tightest budget (Appendix A.2); under the logical convention, which every main-text number uses, it is competitive at every budget. Non-termination also sets a policy's speed (§3.3).

3.3 READING THE ATTENTION WEIGHTS COSTS AT PREFILL AND UNDER BATCHING

One request at a time. An attention-based policy reads the attention weights, one number for every pair of a query token and a cached token. When one request decodes at a time, each step has a single query token, so the weights are small and reading them is cheap; the cost of the step is set by loading the model's weights and launching its kernels (Pope et al., 2023; Dao et al., 2023; Chen, 2026). Measured directly, a decode step that returns the weights takes the same time as a FlashAttention step at every cache length up to our largest budget (Table 33, Appendix F.1). An end-to-end run of every policy at equal load agrees. The attention-based policies are among the fastest, and a policy's speed follows the work it does between forward passes; EPIKV's query term, one product per compaction, costs nothing measurable per token (Table 34). The time a problem takes follows termination instead. A loop emits tokens until the cap, so the loops of §3.2 cost wall-clock time too (Table 21).

Table 4: Both policies keep their harness accuracy when they evict inside vLLM on its stock kernels. MATH-500 problems 0–99, Llama-8B, greedy; no paired vLLM–harness difference is significant.

	K=1024		K=2048	
	acc. (%)	cap hits	acc. (%)	cap hits
EPIKV^{-Q}, vLLM	68.0	24	74.0	17
EPIKV^{-Q}, harness	66.0	23	73.0	18
EPIKV, vLLM	75.0	17	75.0	15
EPIKV, harness	76.0	16	76.0	18

Prefill and batching. At prefill every prompt token is a query, and under batching every request adds one, so the weights grow with the number of queries times the cache. There the pass that returns them is slower than a FlashAttention pass and runs out of memory at lengths and batch sizes that FlashAttention serves (Figure 1, Appendix F.1). A server runs in this regime, and there a budget matters for a second reason. It holds each request's cache flat as traces grow, and the size of that cache decides how many requests fit (Appendix F.3).

Serving, where no engine exposes the weights. No vLLM decode path exposes the attention weights, so attention-based scoring cannot run there without forking the paged-attention kernel. Both of our policies run there on stock kernels, with no change to the engine beyond Python hooks in the scheduler's block allocator, which tell it which blocks a compacted request still owns (Table 4). The port compacts the paged cache in place; the attention kernels, the paged-cache layout and the engine loop are untouched, and no CUDA is written.

The port reaches the harness's accuracy. EPIKV's query term is read from vLLM's own fused projection and rotary module, and its ranking differs from the harness's in detail (Appendix F.2). The cost is small: scoring inside the engine takes a modest share of decode throughput at every concurrency we ran, with negligible tracker state (Table 35); capping the cache takes a similar share while KV memory is plentiful, and once the uncapped engine's KV pool is full the two engines decode at about the same rate (Figure 6, Appendix F.2).

4 Related Work

Attention-based eviction. Most eviction policies rank tokens by attention weight, whether they target long prompts (Zhang et al., 2023; Xiao et al., 2024; Li et al., 2024a; Cai et al., 2025b; Liu et al., 2025; Goel et al., 2025; Gu et al., 2025; Zhou et al., 2024) or the long generations of reasoning models (Ramachandran et al., 2026; Hu et al., 2025; Cai et al., 2025a; Su et al., 2026; Dong et al., 2026). Table 1 lists what each policy we compare against reads and the kernel it needs. Reading attention weights requires the attention matrix that fused kernels such as FlashAttention avoid forming (Dao et al., 2022; Dao, 2024), so every attention-based baseline runs on the eager kernel or its own.

Eviction-induced loops. Eviction can send a reasoning model into repetition. RaaS reports it for H2O (Hu et al., 2025), and VaSE attributes it to evicting large-magnitude value states and protects them (Chang et al., 2026). These works establish that the loops occur; the split of §3.2 shows, further, that at tight budgets they account for the accuracy lost.

Signals without the attention matrix. Attention-free scores exist: the ℓ_2 norm of the cached key (Devoto et al., 2024), block-level statistics inside vLLM's paged cache (Chitty-Venkata et al., 2025), and lag-relative KV statistics (Liang et al., 2025). All are fixed when a token is cached and blind to the model's current queries. LagKV keeps a fixed share of each partition, so its cache grows with the sequence; our Lag-KV brings its normalization into a fixed-budget policy (§2.4) and represents this family in our comparison; it trails the query-aware policies at tight budgets (Table 2). Outside eviction, speculative decoding already drafts from hidden states because they carry more predictive structure than token sequences (Li et al., 2024b). Within eviction, no prior decode-time policy for reasoning traces scores tokens from the residual stream or recomputes the query from it.

Orthogonal directions. Retrieval keeps every token and fetches a subset per step (Tang et al., 2024; Liu et al., 2026), SideQuest trains the model to delete stale tool output (Kariyappa & Suh, 2026), and quantization lowers the precision of what is kept (Hooper et al., 2024; Sharma et al., 2025); each composes with eviction.

5 Discussion

Band A composes content; Band B predicts the next token. Band A coincides with the mid-network layers that interpretability work identifies as the site of factual recall and feature composition (Meng et al., 2022; 2023; Lad et al., 2025), so a large hidden-state change there marks a token at which the model retrieves or composes content. Band B sits where the residual stream turns toward the output distribution (Geva et al., 2021; Lad et al., 2025). A large change there accompanies fluent, predictable continuation, and the tokens it marks are the ones the model can do without. Subtracting the bands uses both signals at once; at eviction time the difference and Band A alone are within noise (Appendix B.4). The bands also move with the task, deeper on competition mathematics than on GSM8K (Appendix B.6), and the residual stream drifts with position over a long generation (§2.2). Any score read from the stream inherits that drift; a causal rolling z-score is a cheap remedy.

Report loops and wrong answers separately. Accuracy alone hides how a policy fails. The same loss can come from loops or from wrong answers, and the two call for different fixes. EpiKV^{-Q}'s loss comes from loops alone, for instance, and ThinKV's from both (Table 3). The split costs nothing to compute from the per-problem records every harness already writes, and it tells a designer which lever moved. We recommend that eviction papers report it beside accuracy, together with the generation cap, since the cap decides how a loop is counted.

6 Conclusion

At tight budgets, the accuracy that eviction costs a reasoning model goes to generations that never stop. Keeping the tokens the model's recent queries point to lets them stop, and EpiKV identifies those tokens from the cached keys and the hidden states the forward pass already returns. Because it needs neither the attention matrix nor a custom kernel, EpiKV runs where reasoning models are served: inside a paged engine, on FlashAttention, at the tight budgets where eviction matters most. The split of errors that located the problem also gives the next policies their target, a cache that keeps every generation able to stop.

Reproducibility statement

The harness, every eviction policy including the adapted baselines (Appendix A.4), the counterfactual labels under every filler (§2.2), the marked traces behind Figure 1 (left), the kernel-cost microbenchmarks (Appendix F.1) and the vLLM integration are released with the code. The experimental setup at the start of §3 and §2.3 give every setting of EpiKV and its ablations, and Appendix A.1 lists the hardware and library stack behind each table. Greedy decoding makes each configuration deterministic on its platform, so a rerun on the same stack reproduces every number exactly, and every comparison between policies is made within one stack (Appendix A.1).

AI use statement

In this work, we used generative AI tools as coding and writing assistants: to implement and run experiment and analysis code, and to edit prose, under the authors' direction. The research questions, the method and the claims are the authors'. The authors reviewed and tested all AI-assisted code, checked every reported number against the underlying result files, and take full responsibility for the content of this paper.

References

Mislav Balunović, Jasper Dekoninck, Ivo Petrov, Nikola Jovanović, and Martin Vechev. MathArena: Evaluating LLMs on uncontaminated math competitions. *arXiv preprint arXiv:2505.23281*, 2025.

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 *Second Conference on Language Modeling (COLM)*, 2025b.

Ting-Yun Chang, Harvey Yiyun Fu, Deqing Fu, Chenghao Yang, Jesse Thomason, and Robin Jia. Value-aware stochastic KV cache eviction for reasoning models. *arXiv preprint arXiv:2606.03928*, 2026.

Josef Chen. Memory-bound but not bandwidth-limited: The physical AI inference gap in batch-1 LLM decode. *arXiv preprint arXiv:2605.30571*, 2026.

Krishna Teja Chitty-Venkata, Jie Ye, Xian-He Sun, Anthony Kougkas, Murali Emani, Venkatram Vishwanath, and Bogdan Nicolae. PagedEviction: Structured block-wise KV cache pruning for efficient large language model inference. *arXiv preprint arXiv:2509.04377*, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, volume 2024, pp. 35549–35562, 2024.

Tri Dao, Dan 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)*, volume 35, pp. 16344–16359, 2022.

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. PyTorch blog, https://pytorch.org/blog/flash-decoding/, 2023.

Alessio Devoto, Yu Zhao, Simone Scardapane, and Pasquale Minervini. A simple and effective L_2 norm-based strategy for KV cache compression. *arXiv preprint arXiv:2406.11430*, 2024.

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 *Forty-third International Conference on Machine Learning*, 2026.

Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5484–5495, 2021.

Raghavv Goel, Junyoung Park, Mukul Gagrani, Dalton Jones, Matthew Morse, Harper Langston, Mingu Lee, and Chris Lott. CAOTE: KV cache selection for LLMs via attention output error-based token eviction. *arXiv preprint arXiv:2504.14051*, 2025.

Yifeng Gu, Zicong Jiang, Jianxiu Jin, Kailing Guo, Ziyang Zhang, and Xiangmin Xu. AhaKV: Adaptive holistic attention-driven KV cache eviction for efficient inference of large language models. *arXiv preprint arXiv:2506.03762*, 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(8081):633–638, 2025.

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

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 1270–1303, 2024.

Junhao Hu, Wenrui Huang, Weidong Wang, Zhenwen Li, Tiancheng Hu, Zhixia Liu, Xusheng Chen, Tao Xie, and Yizhou Shan. RaaS: Reasoning-aware attention sparsity for efficient LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 2577–2590, Vienna, Austria, 2025. Association for Computational Linguistics.

Sarthak Jain and Byron C. Wallace. Attention is not explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3543–3556, 2019.

Sanjay Kariyappa and G. Edward Suh. SideQuest: Model-driven KV cache management for long-horizon agentic reasoning. *arXiv preprint arXiv:2602.22603*, 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 *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023.

Vedang Lad, Jin Hwa Lee, Wes Gurnee, and Max Tegmark. Remarkable robustness of LLMs: Stages of inference? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 22947–22970, 2024a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning*, pp. 28935–28948. PMLR, 2024b.

Manlai Liang, JiaMing Zhang, Xiong Li, and Jinlong Li. LagKV: Lag-relative information of the KV cache tells which tokens are important. *arXiv preprint arXiv:2504.04704*, 2025.

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.

Guangda Liu, Chengwei Li, Zhenyu Ning, Jing Lin, Yiwu Yao, Danning Ke, Minyi Guo, and Jieru Zhao. FreeKV: Boosting KV cache retrieval for efficient LLM inference. In *The Fourteenth International Conference on Learning Representations*, 2026.

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)*, volume 38, pp. 28728–28778, 2025.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 17359–17372, 2022.

Kevin Meng, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. Mass-editing memory in a transformer. In *International Conference on Learning Representations (ICLR)*, 2023.

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

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

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level Google-proof Q&A benchmark. In *First Conference on Language Modeling (COLM)*, 2024.

Akshat Sharma, Hangliang Ding, Jianping Li, Neel Dani, and Minjia Zhang. MiniKV: Pushing the limits of 2-bit KV cache via compression and system co-design for efficient long context inference. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 18506–18523, Vienna, Austria, 2025. Association for Computational Linguistics.

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

Yi Su, Zhenxu Tian, Dan Qiao, Yuechi Zhou, Juntao Li, and Min Zhang. LongFlow: Efficient KV cache compression for reasoning models. *arXiv preprint arXiv:2603.11504*, 2026.

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

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

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

Xiabin Zhou, Wenbin Wang, Minyan Zeng, Jiaxian Guo, Xuebo Liu, Li Shen, Min Zhang, and Liang Ding. DynamicKV: Task-aware adaptive KV cache compression for long context LLMs. *arXiv preprint arXiv:2412.14838*, 2024.

Appendix contents

Limitations and future work **14**

A Harness, platforms and baselines **14**

A.1 Harness and platforms 14

A.2 The positional convention decides whether H2O collapses 15

A.3 What each policy always keeps 15

A.4 How R-KV and LongFlow were adapted 16

A.5 Our ThinKV implementation, and a stateless variant 16

B The epiphany score **17**

B.1 Counterfactual labels and the stop-token filler 17

B.2 Per-layer correlations 17

B.3 Statistical power of the correlations 19

B.4 The window and the bands are not finely tuned, and Band B is dispensable 19

B.5 The two-band anatomy on Qwen-7B 21

B.6 On GSM8K the bands move 21

C The policy **21**

C.1 What the policy keeps 21

C.2 Attention does not make a measurably better segment classifier 24

C.3 The query term and the fill help only together 24

C.4 Compacting less often matches compacting every step 25

D Full accuracy results **26**

E Termination **28**

E.1 A generation that finds the answer and cannot stop 28

E.2 Most loops are not an artifact of the length cap 29

E.3 Protecting large value states changes neither loops nor accuracy 29

E.4 EpiKV loops at the same rate on both models 30

F Kernel cost and serving **30**

F.1 Where reading the attention weights costs 30

F.2 EpiKV inside vLLM 32

F.3 How many requests fit, with and without a budget 33

LIMITATIONS

Both models are distilled from DeepSeek-R1. Every result uses greedy decoding, which keeps every comparison between policies exactly paired. On AIME and GPQA-Diamond the uncapped model itself fails to stop on a large share of problems, so those benchmarks bound what any policy can recover (Tables 25 and 26). Kernel costs and serving throughput are measured on a single GPU.

Future work. The query term reads one layer from a short window of recent queries; a deeper read, or a relevance term inside the tiering as well, is the natural next step toward the uncapped model at the tightest budget. Beyond occlusion, in-distribution replacement and gradient-based attribution would test whether counterfactual importance is a stable target; the labels we release under every filler are the baseline for that comparison.

A HARNESS, PLATFORMS AND BASELINES

A.1 HARNESS AND PLATFORMS

Eviction is applied through the HuggingFace `DynamicCache`. Retained tensors are written back into the live cache's own per-layer slots rather than into a freshly constructed cache; a policy that evicts at every step would otherwise hold two copies of the cache, a cost that grows with the cache. `_seen_tokens` is set to the retained length, without which the model builds a causal mask one position too long. Keep-masks are placed on each tensor's own device, so multi-GPU `device_map="auto"` runs are correct. FlashAttention runs use a single GPU. The no-eviction baseline is run once and reused across budgets. Table 5 lists where each exhibit ran. Tables 6 and 21 and the A100 blocks of Table 16 count τ over the scored cache: a compaction fires when the number of scored tokens reaches a multiple of τ, which without the fill compacts every few hundred decode steps. Their $\tau>1$ cells compare only within their own table.

Table 5: Where each exhibit ran. [a]With torch 2.7.0 and FlashAttention-2 2.8.3. [b]Its L40S blocks; its A100 blocks ran on the accuracy-grid platform. The accuracy grids ran several cells per card, so their wall-clock times are load-confounded and are not reported; every timing comes from a run with one process per card. Any other table states its platform in its caption.

Exhibit	GPU	Library	Load
Accuracy grids and the tables built from them (Tables 2, 3, 23, 24, 25), component and interval cells (Tables 20, 22), the loop of Appendix E.1	A100	transformers 4.57.1[a]	two to four cells per card
Equal-load timing (Table 34)	A100	transformers 4.57.1[a]	one process per card
Kernel costs (Tables 33, 32)	A100, 40 GB	transformers 4.57.1[a]	one exclusive card
Prefill memory (Figure 1, right)	A100, 80 GB	transformers 5.2.0	one process
Eviction frequency and band ablations (Tables 21, 16[b]), the fill ablation (Table 19), the segment classifiers (Table 18), the right column of Table 6	L40S	transformers 5.2.0	one process per card
vLLM eviction and hook cost (Tables 4, 35)	A100	vLLM 0.11.2	one engine per card
vLLM throughput curve (Figure 6)	H100	vLLM 0.11.2	one engine

A library version changes greedy traces, and with them accuracy on a fixed problem set, by a few problems in a hundred (Table 6). The two stacks return the same hidden states up to numerics and tokenize the prompt to within a token, so the difference is numerical, not an indexing difference. The main tables use the stack on which our policies score lower, with every baseline measured on the same stack, so every comparison in the paper is made within one platform.

Table 6: The main tables use the stack on which our policies score lower. MATH-500 accuracy (%) on the same 100 problems at K=1024 under two inference stacks; the left column is the platform of Table 2. The stacks return the same hidden states up to numerics (per-layer norms within 4% from layer 3 on, last-token cosines 0.92–0.99) and tokenize the prompt to 29 and 28 tokens. The τ=128 row counts τ over the scored cache.

	transformers 4.57.1, A100	transformers 5.2.0, L40S
EPIKV^{-Q} (τ=128, scored-cache rule)	68.0	74.0
EPIKV^{-Q} (τ=1)	66.0	70.0
raw score	66.0	71.0
RaaS	77.0	77.0
H2O	72.0	75.0
none	77.0	77.0

A.2 THE POSITIONAL CONVENTION DECIDES WHETHER H2O COLLAPSES

Where a retained token sits after eviction is a choice the harness makes (§2.1). Under the rewound convention the survivors are re-embedded at the compacted cache's physical slots, so their rotary positions no longer match the distances between them in the text. H2O then collapses on reasoning traces (Table 7), and RaaS reports the same collapse for H2O (Hu et al., 2025). Under the logical convention, which every table not labeled rewound uses, the collapse does not occur and H2O is a competitive baseline at every budget.

Table 7: H2O's collapse on reasoning traces comes from the positional convention, not from the policy. MATH-500 accuracy (%) of H2O on Llama-8B, problems 0–99, eager attention, 8,192-token cap, with the survivors re-embedded at the compacted cache's physical slots (rewound; another platform of this harness) and kept at their original positions (logical; the same shard of Table 2). Cap hits are generations that reached the cap; empty are generations of no tokens. Under rewound positions every generation at K=512 loops to the cap and most at K=1024 stop empty; under logical positions neither failure occurs. The two columns ran on different platforms and are not paired.

	rewound positions			logical positions		
K	acc	cap hits	empty	acc	cap hits	empty
512	1.0	100	0	61.0	38	0
1024	5.0	0	93	72.0	21	0
2048	49.0	0	48	78.0	14	0
4096	67.0	0	27	75.0	19	0

A.3 WHAT EACH POLICY ALWAYS KEEPS

The policies differ in which structural tokens they exempt from scoring and in whether a compaction fills the budget (Table 8). H2O keeps attention sinks and a recency window; RaaS and our policies keep the whole prompt and a recency window; ThinKV and LongFlow keep a recency window only, and R-KV adds its last eight observed tokens. RaaS runs its token-level algorithm (Hu et al., 2025, Algorithm 1). Two policies can hold fewer than K tokens after a compaction: ThinKV, whose per-segment budgets need not sum to K, and EPIKV^{-Q}, whose tier caps bind before the budget does. The recency window is $\min(128, K/4)$ throughout.

These conventions also vary across the literature, which matters when comparing published numbers. In LongFlow's harness, for instance, H2O receives a recency window of half the budget and the prompt is compressed once at prefill, a larger structural allocation than the one used here; the released evaluation code also applies a repetition penalty to every method, which suppresses the looping that eviction induces on reasoning traces (§3.2). Published H2O figures on reasoning traces therefore depend on the structural allocation, the decoding penalty and the positional convention (Appendix A.2) as much as on the policy.

Table 8: The policies keep different structural tokens, and two of them can hold fewer than K. "Always kept" is exempt from scoring; "Holds" is the cache size after a compaction. ThinKV's segment types are reasoning, execution and transition; R-KV keeps its last eight observed tokens.

Method	Always kept	Ranked by	Holds
H2O	4 sinks, recency	cumulative attention	K
ThinKV	recency	segment type	$\leq K$
RaaS	prompt, recency	attention timestamp	K
R-KV	8 observed, recency	importance minus redundancy	K
LongFlow	recency	attention $\times \|v\|_1$	K
EPIKV	prompt, recency	tiers, query term, fill	K
EPIKV$^{-\mathrm{Q}}$	prompt, recency	tiers, $s(t)$	$\leq K$
EPIKV$^{-\mathrm{QS}}$, raw score	prompt, recency	$s(t)$ or its raw form	K
KV-key, Lag-KV	prompt, recency	key or key-and-value variance	K

A.4 How R-KV and LongFlow were adapted

R-KV (Cai et al., 2025a) and LongFlow (Su et al., 2026) are implemented in our harness from their papers' equations and reported at n=495 on the platform of Table 2. Two adaptations make them comparable with the other rows. First, R-KV selects tokens per head and LongFlow per layer, whereas our harness applies one keep-mask to the whole cache; we therefore average each method's scores over heads and layers before selection. Second, R-KV's importance and redundancy terms are standardized at each compression step before they are mixed, since its published mixing weights otherwise leave the redundancy term numerically inert under this averaging; its 128-step compression interval is kept. LongFlow's score, attention $\times \|v\|_1$, is computed eagerly rather than through its released Triton kernel, which changes the speed of the computation and not the tokens it selects. The released code documents both adaptations.

A.5 Our ThinKV implementation, and a stateless variant

Every ThinKV number in this paper comes from our implementation: segment entropies classified by a Gaussian kernel density with Silverman's robust bandwidth, boundaries at the two deepest interior minima, and reclassification every 128 steps. We also evaluated a simpler variant that splits the entropies at tertiles and reclassifies at every decode step (Table 9). Under the rewound-position protocol of Appendix A.2, it scores within noise of that implementation at every budget, so in this harness neither the density fit nor the refresh interval explains ThinKV's place. We report the variant here and not as a baseline, because it departs further from ThinKV. The density engages only when a cache holds enough segments for a multimodal entropy distribution, which at 128-token segments means $K \geq 2048$; below that the implementation falls back to tertiles, and the two variants differ only in the refresh interval.

Table 9: A stateless variant of ThinKV scores within noise of our ThinKV implementation. MATH-500 accuracy (%) on Llama-8B, n=100 per cell, whose Wilson interval spans roughly $\pm$9 pp, under the rewound-position protocol of Appendix A.2 (not comparable to Table 2). The variant splits segment entropies at tertiles and reclassifies at every decode step; our implementation fits a density and reclassifies every 128 steps. Below K=2048 the two differ only in the refresh interval.

Implementation	512	1024	2048	4096
ThinKV (KDE, τ=128)	4.0	35.0	52.0	68.0
Stateless tertile variant (ours)	6.0	32.0	58.0	71.0

B THE EPIPHANY SCORE

B.1 COUNTERFACTUAL LABELS AND THE STOP-TOKEN FILLER

Labels come from sliding-window occlusion over the reasoning span of each correctly answered trace. The window is 32 tokens with stride 16, so each interior position is covered by two windows. The reasoning span ends at the </think> token sequence, or failing that at the last \boxed{, or failing that 64 tokens before the end. Each window's tokens are replaced by the tokenizer's pad id, which for this model resolves to the end-of-sequence (EOS) token, since the model defines no pad token. The modified context up to the end of the reasoning span is fed back, and the answer is regenerated greedily with up to 512 new tokens. A position is labeled important if any window covering it changes the answer; prompt positions count as important, and the answer span is not tested. Regeneration is deterministic, so the labels are reproducible.

A labeled trace. On MATH-500 problem 3 ("How many positive whole-number divisors does 196 have?"), 3 of the 62 windows tested change the answer. One covers the plan, the sentence that commits to the method; the other two cover the conclusion. Every other span, the factorization included, can be removed without changing the answer. Important tokens are marked with double brackets (Llama-8B, EOS-filler labels).

```
Okay, so I need to figure out how many positive whole-number divisors
196 has. Hmm, divisors, right? [...] I remember that to find the number
of divisors, I need to look at the prime factorization of the number.
[[ Once I have that, there's some formula or method to calculate the
number of divisors based on the exponents in the prime factors. Let me
try to]] recall how that works.

First, let me find the prime factors of 196. [...]

[...] So, yeah, 9 is[[ the right answer.

**Final Answer**
The number of positive whole-number divisors of 196 is \boxed{9}.
</think>]]
```

Filler controls. An EOS filler injects a stop signal as well as removing content, so the labeling is repeated with three stop-free fillers on matched traces and windows: an untrained pad token (<|finetune_right_pad_id|>), a repeated random in-vocabulary token, and a high-entropy filler that draws a distinct random token per position. Paired over the same 29 traces, the three stop-free fillers agree with one another and each differs from EOS, which attributes about a third of the EOS flips to the stop signal (Table 10). Under the high-entropy filler, across all 60 traces, every Band A layer stays positive and every Band B layer stays negative, at magnitudes two to four times smaller than under EOS; the stop-signal flips account for the difference.

Table 10: A third of the EOS flips are stop semantics, and the anatomy holds without them. Flip rate per occluded window, paired over the same 29 MATH-500 traces (Wilcoxon against EOS; the three stop-free fillers agree pairwise, $p \geq 0.11$), and the mean per-layer ρ over each band under the EOS labels (Table 11) and under the high-entropy filler across all 60 traces. The two rows without band means were not relabeled at the per-layer level.

Filler	flip rate	vs. EOS (p)	Band A mean ρ	Band B mean ρ
EOS (pad id)	0.176	–	$+0.077$	-0.200
untrained pad token	0.115	0.023–0.052	–	–
repeated random token	0.114	0.023–0.052	–	–
high-entropy random tokens	0.118	0.023–0.052	$+0.019$	-0.092

B.2 PER-LAYER CORRELATIONS

Table 11 gives the Spearman ρ between $\bar{g}_l$, the hidden-state change at layer l averaged over a 64-token rolling window, and the counterfactual importance labels, for all 32 layers on the two competition-

mathematics datasets under both attention back-ends; it is the data behind Figure 3. Band A (7–13) is positive throughout and Band B (18–25) negative throughout on MATH-500 under both back-ends. The AIME-2024 columns keep each band's sign in the mean, at a sample size that does not support a per-layer claim (Appendix B.3). The last layer flips sign across datasets and is not used. Table 12 sets the two attention-derived signals beside the deployed band layers on the configurations where attention weights exist.

Table 11: Band A is positive and Band B negative at every layer on MATH-500 under both back-ends. Spearman ρ of $\bar{g}_l$ with the importance labels, all 32 layers, Llama-8B, EOS-filler labels; Band A (7–13) and Band B (18–25) rows are in bold.

l	math500	m500-eag	aime24	aime24-eag
0	−0.173	−0.199	−0.208	+0.037
1	−0.172	−0.202	−0.208	−0.052
2	−0.121	−0.151	−0.158	−0.125
3	−0.121	−0.124	−0.150	−0.072
4	−0.153	−0.139	−0.191	−0.052
5	−0.079	−0.058	−0.105	+0.093
6	+0.011	−0.007	−0.048	+0.150
7	+0.079	+0.047	−0.010	+0.156
8	+0.067	+0.078	+0.037	+0.155
9	+0.082	+0.107	+0.054	+0.130
10	+0.112	+0.141	+0.040	+0.120
11	+0.092	+0.144	+0.079	+0.120
12	+0.038	+0.077	+0.076	+0.114
13	+0.071	+0.089	+0.009	+0.065
14	−0.016	+0.006	−0.076	−0.032
15	+0.017	+0.016	−0.201	−0.147
16	+0.008	−0.016	−0.205	−0.165
17	−0.003	−0.005	−0.173	−0.191
18	−0.081	−0.045	−0.129	−0.184
19	−0.147	−0.074	−0.099	−0.208
20	−0.191	−0.097	−0.037	−0.224
21	−0.209	−0.109	−0.011	−0.227
22	−0.223	−0.121	−0.007	−0.217
23	−0.254	−0.151	+0.007	−0.200
24	−0.250	−0.146	+0.016	−0.188
25	−0.243	−0.142	−0.034	−0.187
26	−0.211	−0.107	−0.013	−0.157
27	−0.066	−0.003	−0.030	−0.086
28	−0.053	−0.018	+0.002	−0.059
29	+0.082	+0.049	−0.042	+0.030
30	+0.135	+0.065	−0.035	+0.062
31	+0.220	+0.092	−0.196	−0.022

Table 12: Cumulative attention, the signal H2O ranks by, is a weaker correlate of importance than either deployed band layer. Spearman ρ with the counterfactual importance labels on the two eager configurations, where attention weights exist: H2O's cumulative attention, attention entropy, and the hidden-state change at the deployed layers of Band A and Band B and the KV-key variance, each averaged over a 64-token rolling window. Attention entropy is the stronger attention-derived signal and is competitive with the band layers, but it needs the same attention matrix.

	attention-derived		hidden-state change		KV-key
Dataset (eager)	cumulative	entropy	$\bar{g}_{10}$	$\bar{g}_{21}$	variance
MATH-500	+0.050	+0.176	+0.141	−0.109	+0.214
AIME-2024	−0.011	−0.139	+0.120	−0.227	−0.022

B.3 Statistical power of the correlations

Effective sample size counts independent traces rather than token pairs, since tokens within a trace are correlated. Table 13 gives n_{eff} and the resulting standard error of Spearman ρ for each configuration. MATH-500 and GSM8K have enough traces to support a per-layer claim; every AIME configuration has an interval that spans zero. The paper therefore makes the per-layer claim on MATH-500, and in its shifted form on GSM8K; on AIME it reports the sign of each band and makes no per-layer claim.

Table 13: Only MATH-500 and GSM8K have enough traces for a per-layer claim. Effective sample size per configuration and the standard error of Spearman ρ, $1/\sqrt{n_{\text{eff}}}$; n_{eff} counts independent traces, not scored tokens.

Dataset	$n_{\text{eff}} \uparrow$	$\pm$SE (ρ) $\downarrow$
math500	72	0.118
math500-eager	78	0.113
gsm8k-eager	352	0.053
aime2024	11	0.301
aime2024-eager	8	0.354
aime2025/2026	≤ 5	≥ 0.45

B.4 The window and the bands are not finely tuned, and Band B is dispensable

The causal rolling window raises the correlation of every deployed signal with the labels above its raw form, and above an exponential moving average, in every configuration (Table 14). These choices are made on ρ; what matters for deployment is whether they change accuracy. Under the rewound-position protocol of Appendix A.2, moving the band layers by two or changing the window by a factor of four leaves accuracy at $K{=}2048$ identical across all eight variants, with the same prediction on every problem, and moves it by one problem at $K{=}1024$, inside a sampling interval an order of magnitude wider (Table 15).

Table 14: The causal rolling window raises the correlation of every deployed signal above its raw form. $|\rho|$ with the counterfactual importance labels under three smoothings, Llama-8B; the exponential moving average uses $\alpha{=}0.9$, and the rolling window spans 64 tokens. The AIME-2024 rows rest on few traces (Table 13). Pre-RoPE key statistics change $|\rho|$ by at most 0.0005 in any cell and are not used.

Configuration	Signal	raw	EMA	rolling
MATH-500	$\bar{g}_{10}$ (Band A)	0.034	0.084	0.112
	$\bar{g}_{21}$ (Band B)	0.085	0.180	0.209
	KV-key variance	0.288	0.356	0.380
MATH-500 (eager)	$\bar{g}_{10}$ (Band A)	0.041	0.101	0.141
	$\bar{g}_{21}$ (Band B)	0.043	0.089	0.109
	KV-key variance	0.150	0.193	0.214
AIME-2024	$\bar{g}_{10}$ (Band A)	0.003	0.014	0.040
	$\bar{g}_{21}$ (Band B)	0.010	0.010	0.011
	KV-key variance	0.203	0.246	0.259
AIME-2024 (eager)	$\bar{g}_{10}$ (Band A)	0.037	0.094	0.120
	$\bar{g}_{21}$ (Band B)	0.067	0.173	0.227
	KV-key variance	0.014	0.018	0.022

The score is the difference of two band z-scores, $z_{10} - z_{21}$ (§2.2). Table 16 asks what each band contributes, first on EpiKV$^{-\text{QS}}$, a global top-K over the detrended score with no segment tiering, so that nothing but the score changes between rows, and then on EpiKV$^{-\text{Q}}$ itself on two platforms and two models. Every form is within paired noise of the deployed difference: on EpiKV$^{-\text{QS}}$, Band A alone (14 wins / 6 losses), Band B alone and the two alternative layer pairs; on EpiKV$^{-\text{Q}}$, Band A alone in every cell on both platforms and both models. The deployed $z_{10} - z_{21}$, the form the correlations of §2.2 motivate, is therefore kept.

Table 15: The bands and the window are not fitted parameters at this granularity. MATH-500 accuracy (%) with the band layers (a, b) and rolling window W perturbed around the deployed $(10, 21, 64)$, n=100 per cell, measured on the global top-K form (EPIKV$^{-\mathrm{QS}}$) under the rewound-position protocol of Appendix A.2 (Llama-8B, FlashAttention-2; not comparable to Table 2). At K=2048 every variant returns the same prediction on all 100 problems. At K=1024 the eight span 21–22%, a one-problem range inside a Wilson interval of roughly ±9 pp.

Variant	(a, b, W)	K=1024	K=2048
deployed	10, 21, 64	21.0	49.0
A−2	8, 21, 64	21.0	49.0
A+2	12, 21, 64	22.0	49.0
B−2	10, 19, 64	22.0	49.0
B+2	10, 23, 64	21.0	49.0
both −2	8, 19, 64	21.0	49.0
both +2	12, 23, 64	21.0	49.0
window 32	10, 21, 32	22.0	49.0
window 128	10, 21, 128	21.0	49.0

Table 16: Subtracting Band B, dropping it, and moving the bands all change accuracy within paired noise. MATH-500 accuracy (%) on problems 0–99, greedy; in parentheses, wins/losses against the A−B row of the same block. L40S: transformers 5.2.0; A100: the platform of Table 2, with τ counted over the scored cache (Appendix A.1). Rows compare only within a block.

Setting	Score	K=512	K=1024	K=2048	K=4096
EPIKV$^{-\mathrm{QS}}$ *(global top-K), Llama-8B, L40S,* τ=1					
	A−B, layers 10/21 (deployed)	–	62	71	–
	A only (layer 10)	–	70 (14/6)	74 (4/1)	–
	B only (layer 21)	–	63 (5/4)	74 (5/2)	–
	difference, layers 7/12	–	64 (7/5)	72 (4/3)	–
	difference, layers 5/15	–	59 (7/10)	71 (3/3)	–
EPIKV$^{-\mathrm{Q}}$, *Llama-8B, L40S,* τ=1					
	A−B	–	70	77	–
	A only	–	74 (8/4)	75 (3/5)	–
EPIKV$^{-\mathrm{Q}}$, *Llama-8B, A100,* τ=128 *(scored-cache rule)*					
	A−B	65	68	76	73
	A only	64 (5/6)	69 (6/5)	69 (4/11)	76 (4/1)
EPIKV$^{-\mathrm{Q}}$, *Qwen-7B, A100,* τ=128 *(scored-cache rule; layers 20/24)*					
	A−B	67	74	82	83
	A only	65 (5/7)	75 (8/7)	80 (2/4)	84 (2/1)

B.5 THE TWO-BAND ANATOMY ON QWEN-7B

Every layer index above was located on Llama-8B. We ran the same procedure on Qwen-7B (28 layers): eager traces on MATH-500, counterfactual labels, and the per-layer Spearman ρ of the hidden-state change, averaged over a 64-token rolling window, against importance, over 123,599 labeled positions. The shape replicates. A positive band spans layers 14–22 and peaks at layer 20 ($\rho{=}+0.122$), with a negative band directly above it at layers 24–27, strongest at layer 24 (-0.144). Taking the most positive layer of the lower-mid network and the most negative layer of the upper-mid network gives layers 20 and 24, the score layers of every Qwen-7B result in the paper.

Two things differ from Llama-8B. Qwen-7B's strongest negative correlations sit at layers 0–2 (-0.27 to -0.30), twice the magnitude of layer 24, and Llama-8B has no such early band; restricting the choice to the mid network excludes them, so the shared anatomy is the mid-network shape, not the location of the strongest signal. The Qwen labels also use the EOS filler only. Neither point affects the accuracy results, which take the two layers as given. The same procedure finds the same shape on a second model; the layer indices are per model.

B.6 ON GSM8K THE BANDS MOVE

On GSM8K, labeled on 355 correctly answered traces ($n_{\text{eff}}{=}352$), the anatomy shifts (Table 17). The positive signal moves to the early layers, the negative band extends across most of the network, and the last layer turns strongly positive; attention entropy and the KV-key variance both reverse sign relative to MATH-500. On this reading, harder problems carry the load-bearing computation in the mid layers and simpler arithmetic in the early layers. GSM8K traces are also much shorter, however, and are labeled with the EOS filler only, so a trace-length effect is not separated from a difficulty effect. GSM8K therefore serves as a probe of how the anatomy depends on the task, not as an accuracy benchmark.

Table 17: On GSM8K the positive band moves to the early layers, the negative band widens, and two signals reverse sign. Spearman ρ with the counterfactual importance labels, Llama-8B, eager attention, for the deployed band layers, three other layers, attention entropy and the KV-key variance, each averaged over a 64-token rolling window. GSM8K is labeled with the EOS filler only.

Signal	MATH-500 (eager)	GSM8K (eager)
$\bar{g}_0$	-0.199	$+0.181$
$\bar{g}_7$	$+0.047$	$+0.076$
$\bar{g}_{10}$ (Band A, deployed)	$+0.141$	-0.048
$\bar{g}_{15}$	$+0.016$	-0.351
$\bar{g}_{21}$ (Band B, deployed)	-0.109	-0.303
$\bar{g}_{31}$	$+0.092$	$+0.231$
attention entropy	$+0.176$	-0.313
KV-key variance	$+0.214$	-0.261

C THE POLICY

C.1 WHAT THE POLICY KEEPS

The name "epiphany score" claims that high-scoring tokens are the substantive ones: intermediate results, committed steps, the answer forming. Figure 1 (left) shows what the score alone keeps on a real trace; here we show the shipped policy on the same problem. EPIKV at $K{=}1024$, in the configuration of Table 2 ($\tau{=}128$, logical positions), on MATH-500 problem 5 ("A regular hexagon can be divided into six equilateral triangles. If the perimeter of one of the triangles is 21 inches, what is the perimeter of the regular hexagon?"), greedy, Llama-8B: 1,588 tokens generated after a 61-token prompt, five evictions (at tokens 963, 1,091, 1,219, 1,347 and 1,475), and 1,075 tokens retained at the end, 834 of the 1,347 that were ranked and 241 that sat in the recency window of the last eviction or were generated after it and were never ranked. The trace terminated with the correct answer, 42. Retained tokens are bracketed and evicted tokens are not; the never-ranked tail is omitted.

```
[Okay][,][ so][ I][ have] this[ problem] here: A[ regular][ hex][agon]
```

```
[ can][ be][ divided][ into][ six][ equ][ilateral][ triangles][.] [...]
[ Hmm][,][ let][ me] try to figure[ this][ out][ step][ by][ step][.]
[...] It's also[ mentioned] that it can be divided into[ six] equ
[ilateral][ triangles][.] [...] if you draw lines from the[ center] of
[ the] hexagon to each of its vertices, you'll end up with six [...]
[Let][ me][ denote] the[ length] of one[ side] of[ the][ equ]ilateral
[ triangle][ as][ '][s]['.] [...] [3][s][ =][ ][21] [...] [s][ =][ ][7]
[ inches] [...]
[I][ remember][ that] in a regular[ hex][agon][,] the radius[ (]distance
from center to a vertex) is[ equal] to the[ length] of the side of the
hex[agon][.] Is that[ right][?][ Let] me recall.[ Yes], in a regular
hexagon, the radius of the circumscribed circle is equal to the length
of[ each] side[.][ So][,][ the] radius[ is] equal to[ '][s][',][ which]
[ is][ ][7][ inches][.] [...] Per[imeter] of hex[agon][ =] [6][ *][ s]
[ =][ ][6][ *][ ][7][ =][ ][42][ inches][.]
[Wait],[ but][ hold][ on][,][ let][ me][ make] sure[ I][ didn]'t make
[ a][ mistake][.] [...] [then][ ][2][r][ +][ s][ =][ ][21][.][ Since]
[ r][ =][ s][,][ that][ gives] [2][s][ +][ s][ =][ ][3][s][ =][ ][21]
[,][ so][ s][ =][ ][7][.] [...] [so][ the][ perimeter][ is][ ][6][*][7]
[=][42][ inches][.] That seems[ correct][.]
```

The policy keeps the problem's objects and the plan ("regular hexagon", "divided into six equilateral triangles", "step by step", "denote the length of one side . . . as 's' "), and, where the trace turns, the turn itself: "the radius is equal to 's', which is 7 inches", "2r + s = 21. Since r = s, that gives 2s + s = 3s = 21, so s = 7" and "6*7=42 inches" are retained word for word. It drops the discourse around them: "this problem here", "try to figure", "if you draw lines from the center of the hexagon to each of its vertices", "Let me recall", a restated definition of the circumradius, and most function words. Early tokens survive at a high rate as well as late ones (Figure 5), so recency does not explain the selection. Because EPIKV spends the whole budget, it keeps most of the ranked tokens here; EPIKV^{-Q} on its own trace of this problem keeps far fewer and evicts the connective prose more aggressively (Figure 1, left). The marked traces are released with the code, including problems 1 and 4 at the same setting, the EPIKV^{-Q} traces, and an H2O trace of problem 1.

EpiKV, K=1024, MATH-500 problem 5: 1,588 tokens generated, 1,075 kept.

Okay, so I have this problem here: A regular hexagon can be divided into six equilateral triangles. If the perimeter of one of the triangles is 21 inches, what is the perimeter of the regular hexagon? Hmm, let me try to figure this out step by step.First, I know that a regular hexagon has all sides equal and all angles equal. It's also mentioned that it can be divided into six equilateral triangles. I think that means if you draw lines from the center of the hexagon to each of its vertices, you'll end up with six equilateral triangles, each having the same size and shape. So, each of these triangles must be equilateral, which means all their sides are equal.Now, the problem says the perimeter of one of these triangles is 21 inches. Since all sides of an equilateral triangle are equal, if I can find the length of one side, I can figure out the perimeter of the hexagon.Let me denote the length of one side of the equilateral triangle as 's'. The perimeter of the triangle would then be 3 times 's', right? So, if the perimeter is 21 inches, I can set up the equation:3s = 21To find 's', I can divide both sides by 3:s = 21 / 3s = 7 inches Okay, so each side of the equilateral triangle is 7 inches long. Now, since the hexagon is made up of six such triangles, each of these triangles has sides that are sides of the hexagon. Wait, is that correct? Let me visualize this.If the hexagon is divided into six equilateral triangles, each triangle shares sides with the hexagon. So, each side of the hexagon is actually one side of the triangle. That means each side of the hexagon is 7 inches long.But hold on, is that the case? Let me think. If you have a regular hexagon divided into six equilateral triangles, each triangle would have two sides that are sides of the hexagon and one side that is a radius of the circumscribed circle around the hexagon. Hmm, maybe I got that wrong.Wait, no, actually, in a regular hexagon, if you connect the center to all the vertices, you create six equilateral triangles. Each triangle has two sides that are radii of the circumscribed circle and one side that is a side of the hexagon. So, in that case, the sides of the triangles that are part of the hexagon are the sides of the hexagon, and the other two sides are the radii.But in the problem, it says the perimeter of one of the triangles is 21 inches. So, the perimeter of each triangle is the sum of all three sides: two radii and one side of the hexagon. So, if each triangle has a perimeter of 21 inches, that would mean: Perimeter of triangle = 2 * radius + side of hexagon = 21 inches.But wait, I don't know the radius yet. Hmm, maybe I need to relate the radius to the side length of the hexagon.I remember that in a regular hexagon, the radius (distance from center to a vertex) is equal to the length of the side of the hexagon. Is that right? Let me recall. Yes, in a regular hexagon, the radius of the circumscribed circle is equal to the length of each side. So, the radius is equal to 's', which is 7 inches.So, if the radius is 7 inches, then the perimeter of each triangle is 2 * 7 + 7 = 14 + 7 = 21 inches. Oh, that matches the given perimeter. So, that checks out. Therefore, each side of the hexagon is 7 inches. So, the perimeter of the hexagon is 6 times the side length, right? Since a hexagon has six sides.Perimeter of hexagon = 6 * s = 6 * 7 = 42 inches.Wait, but hold on, let me make sure I didn't make a mistake. If each triangle has a perimeter of 21 inches, and each triangle has two radii and one side of the hexagon, then 2r + s = 21. Since r = s, that gives 2s + s = 3s = 21, so s = 7. So, each side of the hexagon is 7 inches, so the perimeter is 6*7=42 inches. That seems correct.Alternatively, if I think of the regular hexagon as being composed of six equilateral triangles, each with side length 's', then each triangle's perimeter is 3s. But wait, if the hexagon is divided into six equilateral triangles, each triangle would have two sides that are sides of the hexagon and one side that is a radius. But in that case, the triangles wouldn't all be equilateral unless the radius is equal to the side length.Wait, maybe I need to clarify this. If the hexagon is divided into six equilateral triangles, then each triangle must have all sides equal. So, if the side of the triangle is 's', then the radius must also be 's'. So, the perimeter of each triangle is 3s = 21, so s = 7. Therefore, the side of the hexagon is 7 inches, so the perimeter is 6*7=42 inches.Yes, that seems consistent. So, I think 42 inches is the perimeter of the hexagon.But just to make sure, let me think of another way. If the hexagon is divided into six equilateral triangles, each triangle has a perimeter of 21 inches. So, each triangle has three sides, each of length 7 inches. So, each side of the triangle is 7 inches, which is the same as the side of the hexagon. Therefore, the perimeter of the hexagon is 6*7=42 inches.Alternatively, if I consider the regular hexagon, it can be divided into six equilateral triangles by drawing lines from the center to each vertex. Each of these triangles is equilateral, so each side of the triangle is equal to the radius, which is equal to the side length of the hexagon. So, each triangle has sides of length 's', so the perimeter is 3s=21, so s=7. Therefore, the perimeter of the hexagon is 6s=42.Yes, that seems consistent. So, I think the perimeter of the hexagon is 42 inches.**Final Answer**The perimeter of the regular hexagon is \boxed{42} inches.</think> A regular hexagon can be divided into six equilateral triangles. Each triangle has a perimeter of 21 inches. First, we find the side length of one of these equilateral triangles. The perimeter of an equilateral triangle is given by \(3s\), where \(s\) is the side length. Therefore, we have:\[3s = 21 \]Solving for \(s\), we get:\[s = \frac{21}{3} = 7 \text{ inches}\]Since each side of the equilateral triangle is 7 inches, and the side length of the hexagon is equal to the side length of the triangles, the perimeter of the hexagon is:\[6 \times 7 = 42 \text{ inches}\]Thus, the perimeter of the regular hexagon is \(\boxed{42}\) inches. <EOS>

Figure 5: What survives is the problem's objects and the steps that carry the argument; what goes is the connective prose. Every generated token of the trace above in reading order: the problem of Figure 1 (left), under EPIKV rather than EPIKV^{-Q}. Blue-highlighted tokens survived eviction by relevance, pale-blue ones were never ranked against a full cache, grey-highlighted ones sat in the recency window, and plain grey text was evicted. Of the 1,347 ranked tokens 834 survive, 228 of the first 400 and 288 of the last 400. The cache held at most 1,151 entries against a budget of 1,024, the overshoot the boundary rule allows (Appendix C.4).

C.2 ATTENTION DOES NOT MAKE A MEASURABLY BETTER SEGMENT CLASSIFIER

The within-segment ranker is fixed to the hidden-state score, and only the signal that classifies segments into tiers varies (Table 18). EPIKV and EPIKV^{-Q} use key variance. The entropy of key variance is the closer attention-free analogue of ThinKV's criterion, since low entropy marks concentrated, reasoning-like segments. Attention sparsity is ThinKV's own criterion; it needs the attention matrix, so it bounds what an attention-free classifier gives up. A fourth row folds attention into the token score instead, scoring each token by its cumulative attention times its hidden-state score with the segments unchanged.

No classifier separates from key variance at either budget (Table 18). Attention sparsity trails it by one to two points, as does the entropy variant; attention folded into the token score is two points below it at K=1024 and four above at K=2048, and no pair is significant. An attention-free classifier therefore gives up nothing measurable.

Table 18: No segment classifier separates from key variance, with or without attention. MATH-500 accuracy (%) with the same within-segment hidden-state ranker and three segment classifiers, and with attention folded into the token score; Llama-8B, problems 0–99, greedy, eager attention, eviction at every step, logical positions, L40S (transformers 5.2.0). In parentheses, wins/losses against key variance on the same problems; no pair is significant (exact McNemar, every $p \geq 0.29$). Not comparable to Table 2.

Segment classifier	FA2	K=1024	K=2048
Key variance (EPIKV^{-Q})	✓	73.0	75.0
Entropy of key variance	✓	71.0 (5/7)	74.0 (5/6)
Attention sparsity (ThinKV's)	✗	72.0 (5/6)	73.0 (3/5)
Attention folded into the token score	✗	71.0 (3/5)	79.0 (6/2)

C.3 THE QUERY TERM AND THE FILL HELP ONLY TOGETHER

EPIKV^{-Q}'s tier caps (64, 32 or 8 tokens per 128-slot segment) can bind before the budget does, so the policy can hold fewer tokens than it may; on long AIME traces at K=8192 it holds under half of the budget. One repair spends the leftover budget on the highest-scoring evicted tokens by the same frozen score. Under the rewound-position protocol of Appendix A.2, Table 19 compares that fill with the unfilled policy on the same 100 problems. The fill is worse at every budget, significantly at the two that bind: it solves no problem at K=512 or 1024 that the unfilled policy misses, and the cap-hit rate rises wherever it adds tokens. More tokens chosen by a frozen score mean fewer generations that stop, in the direction §3.2 predicts. On the pooled AIME set at K=8192, where the budget does not bind, the same fill is neutral, within one problem.

Table 19: The same test under the rewound-position protocol. MATH-500 accuracy (%) and cap hits on problems 0–99, Llama-8B, EPIKV^{-Q} without and with the fill; paired wins/losses of the filled variant against the unfilled one, with the exact McNemar p. L40S, transformers 5.2.0, under the rewound-position protocol of Appendix A.2; not comparable to Table 2.

K	EPIKV^{-Q}	cap-hit	+fill	cap-hit	paired (wins/losses, p)
512	8.0	95	1.0	100	0/7, 0.016
1024	39.0	60	24.0	90	0/15, 0.0001
2048	62.0	32	57.0	49	1/6, 0.125
4096	71.0	24	70.0	26	1/2, 1.0

The same holds under the paper's protocol. On the platform of Table 2 and under the compaction rule of its shipped rows, the fill alone raises the cap-hit count in three of the four cells (from 38 to 43 and from 23 to 38 on Llama-8B at K=512 and 1024, and from 16 to 24 on Qwen-7B at K=1024; unchanged at 35 on Qwen-7B at K=512), costs accuracy in three and helps in none significantly, and the query term alone stays within paired noise of EPIKV^{-Q} in every cell (Table 20). Together they separate from EPIKV^{-Q}, significantly in two of the four cells. The filled tokens are the highest-scoring evicted tokens by a score frozen when they were generated, which is the set the model

Table 20: Neither the query term nor the fill helps reliably on its own; together they make up EPIKV. MATH-500 accuracy (%) on problems 0–99 with the components of EPIKV added to the query-blind EPIKV^{-Q} one at a time, on the platform of Table 2; in parentheses, paired wins/losses against EPIKV^{-Q} on the same problems, with exact McNemar stars (* $p < 0.05$). "–" marks a cell that was not run. w is the query window in tokens. Every row compacts every τ=128 decode steps, the rule of the shipped rows of Table 2, whose first shard these cells share.

	Llama-8B		Qwen-7B	
Variant	K=512	K=1024	K=512	K=1024
EPIKV^{-Q} (no query term, no fill)	59.0	66.0	65.0	78.0
+ fill by $s(t)$, no query term	63.0 (8/4)	62.0 (6/10)	61.0 (4/8)	73.0 (1/6)
+ query term (w=8), no fill	60.0 (6/5)	71.0 (10/5)	65.0 (6/6)	76.0 (2/4)
+ query term (w=1), no fill	–	69.0 (7/4)	–	–
+ query term (w=1) + fill	–	76.0 (15/5)*	–	78.0 (1/1)
EPIKV (w=8 + fill)	66.0 (11/4)	76.0 (12/2)*	73.0 (9/1)*	79.0 (5/4)

has stopped reading; ranked by the query term instead, the same budget helps. EPIKV therefore ships the filled cache ranked by relevance, and EPIKV^{-Q} stays in every table as the ablation.

C.4 COMPACTING LESS OFTEN MATCHES COMPACTING EVERY STEP

EPIKV^{-Q} scores every token as it is generated but compacts the cache only at segment boundaries (§2.3). Table 21 varies how often it compacts, from every step (τ=1) to τ=32 and τ=128 counted over the scored cache (Appendix A.1), on the same 100 problems, one process per card, beside two controls: no eviction, and a score-only run (τ=∞: the score is computed at every step and the cache is never compacted), which separates the cost of the score from the cost of eviction. Accuracy is within paired noise at every budget; the largest paired difference, 6 wins / 2 losses at K=1024, is not significant. The decode rate is flat within 7% across the rows, and the score itself costs nothing measurable, since the score-only run reproduces the no-eviction rate and accuracy. Wall time per problem follows the number of tokens generated, which follows termination.

Table 21: Compacting less often matches compacting every step. EPIKV^{-Q} on Llama-8B, MATH-500 problems 0–99, greedy, 8,192-token cap, one process per L40S card (transformers 5.2.0). W/L: problems solved by the row and not by τ=1, and the reverse; the no-eviction row covers problems 25–99 (n=75); τ>1 is counted over the scored cache (Appendix A.1). Not comparable to Table 2.

K	Policy	Acc.	W/L vs. τ=1	Cap-hit %	Tokens	Tok/s	s/problem
512	no eviction	77	10/2	13	2,974	31.9	93
512	τ=1 (every step)	64	–	37	4,062	31.7	128
512	τ=32	67	5/2	34	3,927	30.5	129
512	τ=128	63	3/4	36	4,023	29.8	135
1024	no eviction	77	5/2	13	2,974	31.9	93
1024	score only (τ=∞)	77	9/2	14	3,056	31.8	96
1024	τ=1 (every step)	70	–	25	3,564	31.0	115
1024	τ=32	69	3/4	26	3,656	30.4	120
1024	τ=128	74	6/2	19	3,206	29.8	108
2048	no eviction	77	3/1	13	2,974	31.9	93
2048	score only (τ=∞)	77	3/3	14	3,056	31.9	96
2048	τ=1 (every step)	77	–	15	3,031	31.0	98
2048	τ=32	77	3/3	15	3,053	30.5	100
2048	τ=128	76	3/4	16	3,103	29.7	104
4096	no eviction	77	1/2	13	2,974	31.9	93
4096	τ=1 (every step)	80	–	14	3,098	31.2	99
4096	τ=32	80	2/2	13	3,034	30.5	99
4096	τ=128	80	3/3	13	3,019	29.8	101

For EPIKV, compacting more often is never better (Table 22). The three short intervals score alike, as do the three long ones, and the split between them is within paired noise at n=100 but consistent in direction. Both policies ship the amortized schedule, which is also the form a paged engine can run (§2.3).

Table 22: Compacting more often does not help EPIKV. MATH-500 accuracy (%) on problems 0–99, Llama-8B, K=1024, on the platform of Table 2, against the compaction interval.

τ (decode steps)	1	8	16	32	64	128
EPIKV accuracy	70	70	71	77	76	76

Two mechanical facts follow from the boundary rule and are visible on the released traces (Appendix C.1). A compaction fires only when the cache is over budget at a boundary, so between boundaries the cache holds up to $K+\tau-1$ entries. An EPIKV$^{-\mathrm{Q}}$ compaction lands below K because the tier caps bind (Appendix C.3), whereas an EPIKV compaction lands at K, so the occupancy saw-tooths around K; the counts for the trace of Figure 5 are in its caption. The overshoot is a fraction of K that shrinks as K grows, against one entry for the per-step form. A strict variant that fires when the cache exceeds $K-\tau+1$ would remove the overshoot; we have not run it.

D FULL ACCURACY RESULTS

This section gives the tables behind §3.1: the members of the EPIKV family on the first MATH-500 shard (Table 23), the paired counts behind the daggers of Table 2 (Table 24), pooled AIME (Table 25) and GPQA-Diamond (Table 26). AIME pools the 2024, 2025 and 2026 sets, n=90; both it and GPQA-Diamond (n=198) run with a 16,384-token cap, so that the uncapped model's own non-termination is measured rather than imposed. ✓ marks the policies that run on FlashAttention.

The family. At n=495 (Table 2) the query term and the fill are worth +8.7, +7.1, +2.6 and −0.8 points on Llama-8B and +3.1, +4.5, −0.2 and −1.6 on Qwen-7B over EPIKV$^{-\mathrm{Q}}$, from the tightest budget to the loosest. Table 23 shows where the rest of the policy earns its place. Segment-tiered allocation carries the tight budgets: EPIKV$^{-\mathrm{Q}}$ leads EPIKV$^{-\mathrm{QS}}$ by 8 and 9 points on Llama-8B and by 11 and 7 on Qwen-7B at K=512 and 1024. On Qwen-7B the z-scored score leads the raw magnitude only once it is allocated by segment, since EPIKV$^{-\mathrm{QS}}$ trails the raw score at K=512 and 2048 and EPIKV$^{-\mathrm{Q}}$ leads it at every budget; on Llama-8B EPIKV$^{-\mathrm{QS}}$ trails and EPIKV$^{-\mathrm{Q}}$ is level.

Table 23: Removing the query term or the segments costs accuracy at the tight budgets. MATH-500 accuracy (%) on problems 0–99, the first of the five shards behind Table 2; the EPIKV, EPIKV$^{-\mathrm{Q}}$ and Lag-KV rows are that shard of its runs, the others were run on this shard only. Rows below EPIKV remove its components in turn; KV-key and Lag-KV score from the cached KV vectors. Bold: best in a column.

		Llama-8B				Qwen-7B			
Method	FA2	512	1024	2048	4096	512	1024	2048	4096
EPIKV	✓	**66.0**	**76.0**	**76.0**	**75.0**	**73.0**	**79.0**	80.0	81.0
EPIKV$^{-\mathrm{Q}}$ (no query term)	✓	59.0	66.0	73.0	71.0	65.0	78.0	80.0	82.0
EPIKV$^{-\mathrm{QS}}$ (no segments either)	✓	51.0	57.0	66.0	73.0	54.0	71.0	74.0	**83.0**
raw score (no detrending)	✓	58.0	66.0	72.0	74.0	57.0	70.0	76.0	81.0
KV-key	✓	58.0	66.0	69.0	72.0	69.0	77.0	**82.0**	80.0
Lag-KV	✓	54.0	65.0	69.0	73.0	40.0	57.0	75.0	81.0

Pooled AIME. No row separates from EPIKV$^{-\mathrm{Q}}$ at either budget on either model once the 40 paired comparisons in Table 25 are corrected (Bonferroni). The smallest uncorrected p is 0.02, for raw score and KV-key over EPIKV$^{-\mathrm{Q}}$ at Llama K=8192 (0 wins / 7 losses) and for EPIKV$^{-\mathrm{Q}}$ over H2O at Qwen K=8192 (15/4). EPIKV$^{-\mathrm{Q}}$ is level with no eviction (Llama 8192: 0/4, p=0.12; Qwen 8192: 13/7, p=0.26), and EPIKV is level with EPIKV$^{-\mathrm{Q}}$ on Llama-8B at 4096 (4/3) and ahead at 8192 (5/0, p=0.06), and slightly below it on Qwen-7B, within noise (5/9, p=0.42; 8/14, p=0.29). A single cell's Wilson 95% interval spans about ±9 points, so the table bounds what any policy can recover at this trace length.

Table 24: Where EPIKV separates from a row and where it does not. Problems won / lost by EPIKV against each row of Table 2 on the same n=495 problems; the stars mark exact McNemar significance (* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$). Against RaaS no Llama-8B budget separates the two policies and every Qwen-7B budget from 1024 upward does; against the query-blind EPIKV$^{-\mathrm{Q}}$ the separation sits at the tight budgets.

	Llama-8B				Qwen-7B			
EPIKV vs.	512	1024	2048	4096	512	1024	2048	4096
none (no eviction)	32/49	28/26	23/15	9/6	10/74***	14/33**	10/15	7/11
EPIKV$^{-\mathrm{Q}}$	62/19***	54/19***	38/25	17/21	41/26	38/16**	20/21	12/20
Lag-KV	80/12***	66/17***	37/14**	10/6	162/6***	110/3***	41/6***	11/8
R-KV	113/7***	64/14***	40/35	34/44	184/6***	85/10***	51/22***	46/25*
ThinKV	204/8***	163/9***	110/17***	59/34*	251/6***	227/6***	145/9***	62/14***
RaaS	30/22	35/33	38/40	34/40	39/35	50/24**	48/24**	48/20***
LongFlow	122/11***	64/25***	39/32	35/43	174/11***	99/16***	55/24***	47/21**
H2O	67/18***	48/30	34/36	34/43	92/25***	67/18***	58/21***	54/20***

Table 25: On pooled AIME no policy separates from another after correction. Accuracy (%) against the cache budget; AIME 2024–2026 pooled, n=90, greedy, 16,384-token cap. *No eviction is budget-independent; it fails to terminate on 61% (Llama-8B) and 62% (Qwen-7B) of these problems. No cell is marked best (paired tests in the text).

		Llama-8B		Qwen-7B	
Method	FA2	4096	8192	4096	8192
none*	✓	26.7	26.7	32.2	32.2
EPIKV	✓	25.6	27.8	27.8	32.2
EPIKV$^{-\mathrm{Q}}$ (τ=128)	✓	24.4	22.2	32.2	38.9
EPIKV$^{-\mathrm{QS}}$	✓	25.6	25.6	31.1	37.8
raw score	✓	30.0	30.0	26.7	35.6
KV-key	✓	27.8	30.0	31.1	38.9
Lag-KV	✓	26.7	28.9	27.8	41.1
RaaS	✗	23.3	25.6	27.8	30.0
H2O	✗	26.7	26.7	25.6	26.7
R-KV	✗	26.7	25.6	26.7	28.9
LongFlow	✗	28.9	25.6	26.7	28.9
ThinKV	✗	14.4	23.3	20.0	23.3

GPQA-Diamond. The 198 questions are four-way multiple choice; the answer is a letter in the boxed span, and the prompt is the harness's MATH prompt with the lettered options in the question. The uncapped model reaches the cap on 76 problems on Llama-8B and 94 on Qwen-7B, and raising the cap to 32,768 on two shards turns none of the twelve cap hits there into an answer (eleven run to the longer cap, one stops wrong). EPIKV is within paired noise of the uncapped model at both budgets on both models, the largest paired difference being 15 wins / 20 losses (Llama-8B, $K{=}1024$, $p{=}0.50$), and within noise of RaaS. On Llama-8B it separates from EPIKV$^{-\mathrm{Q}}$ at both budgets, 28/12 at $K{=}1024$ and 25/10 at $K{=}2048$ ($p{=}0.02$ for both), and 25 and 18 of those wins are generations that reach the cap under EPIKV$^{-\mathrm{Q}}$ and stop under EPIKV. At $K{=}1024$ on Llama-8B every policy reaches the cap more often than the uncapped model (100 to 131 problems against 76) and terminates wrong less often: the accuracy eviction costs there is loops (Table 26).

Table 26: On GPQA-Diamond EPIKV is within noise of the uncapped model. Accuracy (%), cap hits and terminated wrong answers on paired problems ($n{=}198$; Llama-8B at $K{=}2048$, 187), greedy, 16,384-token cap, logical positions, compaction every $\tau{=}128$ decode steps; H200 cards under the library stack of Table 2. ⋆No eviction is budget-independent.

	Llama-8B						Qwen-7B					
	$K{=}1024$			$K{=}2048$			$K{=}1024$			$K{=}2048$		
Method	acc.	cap hits	term. wrong	acc.	cap hits	term. wrong	acc.	cap hits	term. wrong	acc.	cap hits	term. wrong
none⋆	29.8	76	63	31.6	71	57	27.3	94	50	27.3	94	50
EPIKV	27.3	101	43	32.6	71	55	28.3	101	41	28.8	101	40
EPIKV$^{-\mathrm{Q}}$	19.2	131	32	24.6	95	47	27.3	109	35	28.8	96	45
RaaS	29.8	100	39	34.2	69	55	25.8	102	45	26.8	85	61

E TERMINATION

E.1 A GENERATION THAT FINDS THE ANSWER AND CANNOT STOP

MATH-500 problem 10 asks for the least positive multiple of 30 that can be written with only the digits 0 and 2. Both runs below are Llama-8B at $K{=}1024$ under the protocol of Table 2, on the same prompt. Under EPIKV$^{-\mathrm{Q}}$ the model reaches the correct answer about halfway through its generation, restates it 279 times, and then writes the digit 2 until the 8,192-token cap, so the problem counts as a cap hit. Under EPIKV, which adds the query term and the fill, the model reaches the same answer, checks it against the remaining candidates and stops after 3,505 tokens.

EPIKV$^{-\mathrm{Q}}$, the turn into the loop.

```
So, 2220 is the smallest four-digit number. [...] So, is 2220 the
answer? Wait, but 2220 is 2220, but 2220 is 2220, which is 2220,
but 2220 is 2220, which is 2220, which is 2220, which is 2220,
[... "which is 2220" 279 times in all ...] which is 2220. So, the
number is 22222222222222222222222222222222222222222222222222222
[... the digit 2 until the 8,192-token cap ...]
```

EPIKV, the same point in the trace.

```
Four-digit numbers: 2000, 2002, 2020, 2022, 2200, 2202, 2220, 2222.
[...] 2220 is 2220, which is 2220/30 = 74, which is integer. So, 2220
is the first four-digit number that works.

Therefore, 2220 is the smallest positive integer multiple of 30 that
can be written with only the digits 0 and 2.

**Final Answer** [...] \boxed{2220}
```

E.2 MOST LOOPS ARE NOT AN ARTIFACT OF THE LENGTH CAP

Under the protocol of Table 2, doubling the generation cap to 16,384 tokens at $K{=}1024$ on problems 0–99 leaves both policies' accuracy within paired noise: EPIKV^{-Q} gains two problems and EPIKV three, and neither loses any (Table 27). Most generations that reach the shorter cap reach the longer one as well, 18 of 23 under EPIKV^{-Q} and 10 of 16 under EPIKV, so most cap hits are loops that more length would not finish; the rest end between the two caps. The non-termination of §3.2 therefore belongs mostly to the evicted generation and not to the cap. An earlier grid under the rewound-position protocol of Appendix A.2, on a matched 50-problem subset, points the same way (Table 28).

Table 27: Doubling the cap leaves accuracy within noise, and most cap hits persist. MATH-500 problems 0–99, Llama-8B, $K{=}1024$, greedy, the protocol of Table 2. Paired: problems won / lost by the longer cap on the same problems (exact McNemar $p \geq 0.25$).

	cap 8,192		cap 16,384		
Policy	acc. (%)	cap hits	acc. (%)	cap hits	paired
EPIKV^{-Q}	66.0	23	68.0	18	2 / 0
EPIKV	76.0	16	79.0	10	3 / 0

Table 28: The same test under the rewound-position protocol. Accuracy (%) and cap hits on a matched 50-problem MATH-500 subset at $K{=}1024$ under the two caps; rewound-position protocol of Appendix A.2, not comparable to Table 2.

	cap 8,192		cap 16,384	
Policy	acc. (%)	cap hits	acc. (%)	cap hits
EPIKV^{-QS}	28.0	44	22.0	42
EPIKV^{-Q}	38.0	28	40.0	28

E.3 PROTECTING LARGE VALUE STATES CHANGES NEITHER LOOPS NOR ACCURACY

VaSE reports that evicting large-magnitude value states drives reasoning models into loops that never stop (Chang et al., 2026). If that is the mechanism behind the loops of §3.2, exempting those states should lower the cap-hit rate, and a lower cap-hit rate should buy accuracy. We add a reserve to EPIKV^{-Q} that protects the top-r decode positions by value magnitude, over $r{\in}\{8, 16, 32, 64\}$ and three statistics of the cached value vector: its range, its variance and its ℓ_2 norm. All three read the cached values only, so the reserve keeps the policy on FlashAttention. The reserve is applied as a swap rather than an addition: since EPIKV^{-Q}'s tier caps bind before the budget does, reserving extra slots would enlarge the retained set and confound better tokens with more tokens, so we exchange the lowest-scoring retained positions for the highest-magnitude evicted ones and hold the retained count fixed.

Under the paper's protocol (logical positions, FlashAttention-2, the platform of Table 2, compaction every $\tau{=}128$ decode steps), the range reserve with $r{=}8$ matches its unreserved baseline exactly in accuracy at $K{=}1024$ on problems 0–99 and hits the cap on two more problems (Table 29).

Table 29: A value-magnitude reserve leaves accuracy unchanged and the number of loops within noise. MATH-500 problems 0–99, Llama-8B, $K{=}1024$, greedy, 8,192-token cap, logical positions, compaction every $\tau{=}128$ decode steps. Paired: problems won / lost against EPIKV^{-Q} on the same problems (exact McNemar $p{=}1.0$).

	acc. (%)	cap hits	paired
EPIKV^{-Q}	66.0	23	–
+ range reserve, $r{=}8$	66.0	25	6 / 6

The full grid ran under the rewound-position protocol of Appendix A.2 (Table 30). There, at $K{=}1024$ the reserve lowers the cap-hit rate, as VaSE predicts, but the lower rate does not turn into accuracy,

Table 30: A value-magnitude reserve lowers the cap-hit rate at the tight budget and does not raise accuracy. Accuracy and cap-hit rate (%) on a matched n=494 from MATH-500, exact paired McNemar against EPIKV^{-Q}. Absolute values are not comparable to Table 2: this comparison is internally matched but ran under the rewound-position protocol of Appendix A.2.

K=1024	acc	cap-hit	paired
range, r=8	**35.6**	**61.7**	$+32/-24$, p=0.35
EPIKV^{-Q}	34.0	64.6	—
variance, r=8	34.0	64.6	$+29/-29$, p=1.00
K=2048	acc	cap-hit	paired
EPIKV^{-Q}	**57.3**	36.6	—
variance, r=8	56.9	**36.4**	$+20/-22$, p=0.88
range, r=8	55.9	38.3	$+18/-25$, p=0.36

and at K=2048 the reserve raises the cap-hit rate slightly. No pair separates, and the direction of the accuracy difference flips between budgets; the full grid over all twelve variants is negative throughout. EPIKV^{-Q} ships without the reserve, and the variants are in the released code. We tested only the value-magnitude half of VaSE, not its stochastic eviction, so this result concerns the reserve alone and not their method.

E.4 EPIKV LOOPS AT THE SAME RATE ON BOTH MODELS

Table 31 counts, among the problems the uncapped model solves without reaching the cap, how many each policy turns into a loop at K=1024. EPIKV's share is the same on both models; RaaS's rises on Qwen-7B, which is where it loses most problems to the uncapped model (63 against 42, Table 3). The difference is not trace length: without eviction the two models' traces are equally long (median 2,273 and 2,226 tokens on Llama-8B and Qwen-7B, with 88% and 93% longer than the budget).

Table 31: EPIKV's loop rate is the same on both models; RaaS's rises on Qwen-7B. Problems the uncapped model solves without reaching the cap on MATH-500 (n=495) that a policy turns into a generation reaching the 8,192-token cap, at K=1024.

	Llama-8B	Qwen-7B
RaaS	32 of 345 (9%)	56 of 389 (14%)
EPIKV	24 of 345 (7%)	27 of 389 (7%)

F KERNEL COST AND SERVING

F.1 WHERE READING THE ATTENTION WEIGHTS COSTS

Table 32 times one forward pass over a random context at batch size one, eager attention with the weights returned against FlashAttention, keeping only the last token's logits as a real prefill does; it is the time counterpart of the memory curve of Figure 1. Both microbenchmarks here and the decode-step benchmark of Table 33 ran on one exclusive A100-40GB under transformers 4.57.1, each attention mode in its own process; the memory curve of Figure 1 ran on an A100-80GB under transformers 5.2.0. The same measurements on an L40S show the same pattern and are released with the code.

Every policy at equal load. Table 34 runs every policy end to end with one process per card on the same AIME-2024 problems. The decode rate does not split by attention kernel. The eager methods are among the fastest rows on both models, and on Llama-8B the slowest rows are FA2 policies that rescore the whole cache at every step. Most capped caches decode faster than no eviction at this generation length, because the uncapped cache keeps growing; on Llama-8B the exceptions are the two rescoring policies and RaaS. Peak memory in this regime is set by the weights and the cache, not by the method: the rows sit within about a gigabyte of one another, and no eviction is the lowest

Table 32: Reading the attention weights makes a prefill pass 1.3× slower at 1,024 tokens and 1.7× at 2,048, and the pass fails at 4,096 on a 40 GB card; the FlashAttention pass fits 32,768. One forward pass of Llama-8B over a random context of the given length, batch size one, bf16, one A100-40GB, transformers 4.57.1; time is the median of three passes after one warm-up pass and peak memory is the allocator's peak.

	attention weights read (eager)		FlashAttention	
context (tokens)	peak GB	time (s)	peak GB	time (s)
512	15.6	0.06	15.1	0.06
1,024	17.3	0.12	15.3	0.09
2,048	24.1	0.29	15.7	0.17
4,096	OOM	–	16.4	0.32
8,192	OOM	–	17.8	0.67
16,384	OOM	–	20.5	1.47
32,768	OOM	–	26.1	3.57
65,536	OOM	–	OOM	–

Table 33: Returning the attention weights costs nothing when one request decodes at a time up to 8,192 cached tokens, and up to 2.2 times the step time under batching, where it also runs out of memory first. Time of one decode step of Llama-8B against a cache of the given length, eager attention with the weights returned against FlashAttention; one A100-40GB, transformers 4.57.1, each attention mode in its own process, median of five steps after one warm-up.

batch	cache (tokens)	eager (ms)	FlashAttention (ms)	ratio
1	1,024	35.9	37.9	0.95
	2,048	35.9	37.8	0.95
	4,096	35.9	37.9	0.95
	8,192	36.0	37.9	0.95
	16,384	50.4	40.2	1.25
8	1,024	36.9	36.0	1.03
	2,048	50.9	35.9	1.42
	4,096	80.1	36.2	2.21
	8,192	OOM	52.8	–
32	1,024	83.8	39.0	2.15
	2,048	OOM	56.0	–

of them, since a compaction briefly holds two copies of the cache and the eager kernels hold their attention maps. EPIKV's query term costs nothing measurable in decode rate (26.9 against 26.2 tokens/s for EPIKV^{-Q}) and 0.6 GB of memory, the relevance matrix it holds once per compaction.

Table 34: At batch size one the decode rate is set by what a policy does between forward passes, not by the attention kernel. AIME-2024 (30 problems), K=8192, 16,384-token cap, greedy, one process per A100-80GB (transformers 4.57.1, torch 2.7, FlashAttention-2 2.8.3). tok/s: generated tokens over wall time, summed over the problems; cap: problems reaching the cap; GB: mean peak GPU memory. Accuracy on 30 problems (±16 points) only ties each row to its run. The EPIKV row ran on Llama-8B only, on one exclusive A100.

		Llama-8B					Qwen-7B				
Method	Kernel	acc	cap	tok/s	s/prob	GB	acc	cap	tok/s	s/prob	GB
No eviction	FA2	33.3	16	19.9	606	16.5	33.3	19	20.9	572	14.9
EPIKV (τ=128)	FA2	33.3	15	26.9	437	17.6	–	–	–	–	–
EPIKV^{-Q} (τ=128)	FA2	26.7	18	26.2	473	17.0	36.7	17	25.9	453	15.0
EPIKV^{-Q} (τ=1)	FA2	30.0	13	27.2	427	17.0	33.3	18	20.8	566	15.0
raw score	FA2	33.3	16	16.1	738	17.4	36.7	18	27.4	430	15.2
KV-key	FA2	33.3	16	17.3	679	17.4	40.0	16	27.8	419	15.2
RaaS	eager	26.7	19	18.1	688	17.6	40.0	17	19.9	612	15.4
H2O	eager	30.0	18	27.2	456	17.6	43.3	16	29.2	408	15.4
R-KV	eager	26.7	19	29.4	428	17.6	40.0	15	22.4	529	15.4
LongFlow	eager	30.0	17	25.8	486	17.6	36.7	16	29.5	413	15.4
ThinKV	eager	26.7	18	29.7	427	16.9	36.7	18	30.4	408	15.1

F.2 EPIKV INSIDE VLLM

The eviction port. Both policies run inside vLLM 0.11.2 on stock kernels (Table 4). Every τ=128 decode steps, when a request's logical length exceeds K, the port scores the request's cached positions, compacts its paged cache in place, and tells the scheduler's block allocator which blocks the request still owns; the attention kernels and the engine loop are unchanged. The port sends the harness's exact prompt token ids, system instruction included. At the first compaction of MATH-500 problem 1 we saved, from both engines, the Band-A hidden states, the recomputed queries, the cached keys and the query-term relevance at layer 11. The cached keys match to within 1% (relative ℓ_1 difference 0.8%, cosine 0.99997), the hidden states and query norms to within 3 and 6%, and the relevance vectors have cosine 0.993. Their rankings agree only in part, however: the rank correlation is 0.38, and 65% of the 512 most relevant positions are shared. The two engines therefore keep partly different tokens, and they reach the same accuracy within paired noise.

The cost of the score. We register forward hooks on layers 10 and 21 of vLLM 0.11.2's decode path (A100-80GB, 500-token prompts, 512 decode tokens, three runs per configuration) and compute the rolling detrended score online. Under continuous batching the per-layer hidden-state tensor concatenates all sequences in the step, so per-sequence changes are recovered exactly from the scheduler's `query_start_loc` offsets. The hooked values match HuggingFace's `output_hidden_states`, and computing the score online matches computing it offline from the same saved values (Table 35).

Table 35: Computing the score online inside vLLM costs a modest share of decode throughput at every concurrency. Isolated by a paired comparison, the engine with and without the hooks back to back, three runs each. Prefill cost is at most 1.7%; the tracker state is a constant of about 9 MB. The hooked deltas match HuggingFace's within 1.1% relative error, the bf16 accumulation floor, and offline recomputation reproduces the live scores to 10^{-6}. vLLM's own eager-versus-CUDA-graph gap is 7–39%, shrinking with concurrency.

Concurrency	1	8	32	64
Decode-throughput cost	13.9%	9.2%	7.3%	11.1%

Three points of deployment context. First, hooks require eager execution, and a CUDA-graph-compatible integration would fold the score into the graph (Table 35 gives the eager-versus-graph gap). Second, the claim in §3.3 that no vLLM decode path exposes attention weights rests on a scan of the installed source: the sole `output_attentions` occurrence is in a legacy vision encoder,

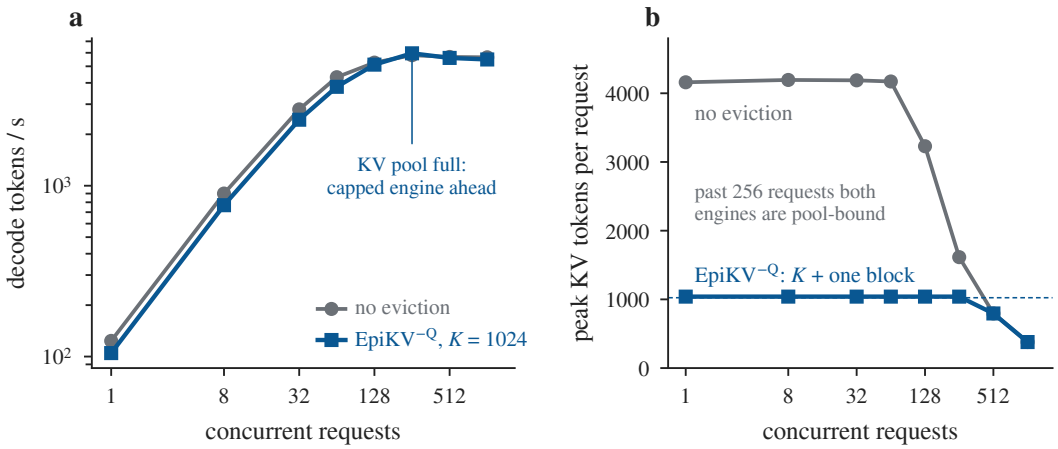

Figure 6: Capping the cache costs a modest share of decode throughput while memory is plentiful and nothing once it is not. The same vLLM engine with and without EPIKV^{-Q} eviction at K=1024, one H100, every request decoding 4,096 tokens. **(a)** Decode tokens per second; from 128 concurrent requests the uncapped engine runs out of KV memory and queues. **(b)** Peak KV-cache tokens held per request: K plus one block for the capped engine; the uncapped value falls once requests wait for blocks.

`model_executor/models/swin.py`. Third, at the 4,096-token decode length of Figure 6 the capped engine's own pool binds only past 256 requests; the throughput case for capping is at the 16,384- to 65,536-token lengths of Figure 1, which we did not run inside vLLM.

F.3 How many requests fit, with and without a budget

Figure 7 projects, from the per-token KV-cache size, the maximum number of concurrent requests that fit on an 80 GB GPU with and without a budget; the measured counterpart is Figure 6. For Llama-8B (32 layers, 8 KV heads, head dimension 128, 16-bit entries) the KV cache costs $2 \times 32 \times 8 \times 128 \times 2$ bytes $= 0.125$ MiB per token. At a 32,768-token context, a full (no-eviction) cache is therefore 4 GiB per request, whereas a cache capped at a 2,048-token budget is 0.25 GiB. With $\approx$56 GiB of the 80 GB available for KV after model weights and runtime workspace, this yields $\lfloor 56/4 \rfloor = 14$ concurrent requests without eviction versus $\lfloor 56/0.25 \rfloor = 224$ with the capped cache, a 16$\times$ ratio equal to the context-to-budget ratio.

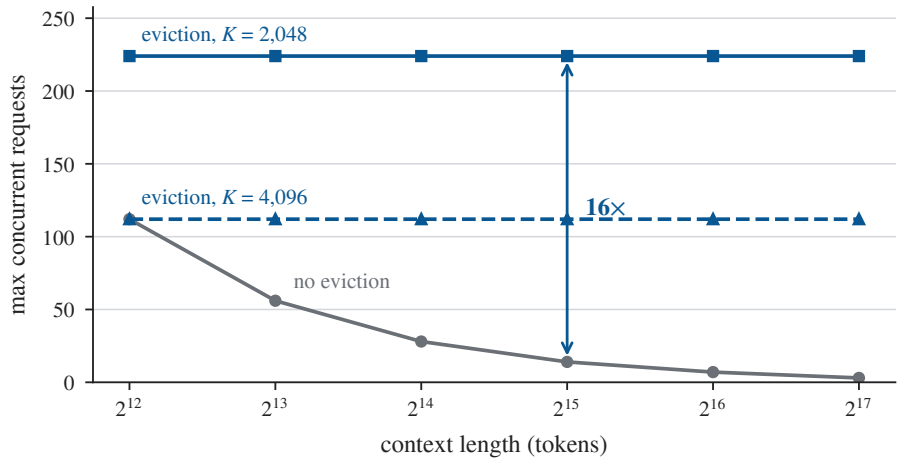

Figure 7: A fixed cache budget holds the number of concurrent requests flat as traces grow; without eviction it falls with the context length. Maximum concurrent requests on an 80 GB GPU against context length, from the KV arithmetic above; the marked gap is the context-to-budget ratio.