RESCUE: GATED CORRECTION FOR KV CACHE EVICTION

Anonymous authors
Paper under double-blind review

ABSTRACT

KV cache eviction is essential for efficient long-context LLM inference. Existing policies score KV entries from evidence already visible at the eviction point, and share a blind spot: at an aggressive budget each misses most of what the continuation will attend to, and policies reading the same observation window fail on overlapping entries rather than independent ones. We propose RESCUE, which treats future-aware eviction as correcting a deployed policy rather than replacing it and then *verifying* the correction against the dense model before it is applied. A policy-conditioned residual scorer is supervised only on the entries its base policy missed, so it never relearns what that policy already ranks correctly; a verification step then decodes a probe from the uncompressed cache immediately after prefill and adopts the corrected cache only if it keeps the output distribution closer to the dense model's. Verification is the part that carries the result: applied unconditionally the correction is harmful on almost a quarter of the evaluated cells. Across all 80 cells of 16 LongBench tasks $\times$ five base policies on Llama-3.1-8B, RESCUE improves the average score of every policy it augments, by $+1.28$ points (95% CI $[+0.51, +2.16]$), and cuts the cells it harms from 18 to 3; the average stays positive on Mistral-7B ($+1.08$) and Qwen3-8B ($+0.43$), with the gain concentrated on the weakest base policy and falling towards zero for the strongest as the budget widens. RESCUE targets evict-once-after-prefill inference and buys that safety with one dense decoding step.

Code, trained scorers and every reported number: `https://anonymous.4open.science/r/RESCUE-54E0`.

1 INTRODUCTION

The KV cache grows linearly with sequence length and batch size (Nawrot et al., 2024) — a 32K-token context with Llama-3.1-70B-Instruct (Grattafiori et al., 2024) needs roughly 10 GB for one sequence — and is re-read at every decoding step (Cai et al., 2025b), so its size drives bandwidth and latency as well as capacity (Behnam et al., 2025). Eviction retains only the entries likely to matter later, under a fixed budget (Xiao et al., 2024; Liu et al., 2023). Like other evict-once methods (Wang et al., 2025), RESCUE reduces the *decode-time* cache, not the prefill peak.

Existing methods rely on one of two importance signals. *Observed importance* uses information already available at the eviction point, such as recent or accumulated attention (Feng et al., 2025; Oren et al., 2024; Qin et al., 2025); *predicted future importance* estimates how important an entry will become for a response not yet generated (Tang et al., 2024). We show these are not two estimates of the same quantity: at a matched budget each recovers oracle-critical entries the other misses entirely — 7.1% by the observed signal alone, 24.6% by the predicted signal alone. The 24.6% is what that same policy discards and a correction can still reach, and it is what RESCUE targets. Blending them under a fixed budget barely beats the predicted signal alone, and oracle coverage does not settle the downstream question either: augmenting an observed policy beats replacing it with either future-aware method (Appendix A.1). Observed policies sharing the same local evidence, meanwhile, fail on overlapping entries more often than independent failures would produce, modestly but systematically and most among policies reading the same observation window, so refining the scoring rule alone is unlikely to reach dependencies not yet visible at eviction.

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.

This motivates a different view: a future predictor need not replace the existing policy. Since an observed policy already identifies many important entries reliably, the objective is to recover only the future-critical entries it misses and, since a learned correction is not uniformly beneficial, to decide when to apply it. We propose RESCUE. It defines a *rescue set* as the difference between the oracle future-attention selection and what the base policy retains, learns to predict that residual from prefill features, and at inference decodes one probe token from the dense cache to decide per document whether the corrected cache stays closer to the dense model's output distribution. Over all 80 (base, task) cells of 16 LongBench tasks (Bai et al., 2024) it improves every base policy it augments on Llama-3.1, and the average on all three backbones.

Our contributions are twofold. The first is the framework: a correction conditioned on the deployed policy, verified against the dense model by one probe token before it is applied. Neither half stands alone — unconditionally the correction harms 18 of 80 cells, and the same verifier given two pre-existing caches instead of a learned correction reaches $+0.50$ against $+1.71$ — so the pair is what is contributed. The second is a measurement of where verification carries the result and where the probe's own attention already answers the question, separating the tasks a learned correction is needed for from the ones it is not (Appendix N.1).

Training on the residual rather than on the full future is a design choice, not a third contribution: against a policy-agnostic target it halves the cells it harms, 3 against 7, and gains $+0.22$ over the 80 cells, which one pair of checkpoints cannot resolve ($p = 0.148$), though retraining both under three seeds separates them where we could repeat it ($+2.10$ against $+1.51$, Appendix P).

2 Background

2.1 KV Cache Eviction

Transformer LLMs (Brown et al., 2020) cache the keys $K = \{k_1, \ldots, k_N\}$ and values $V = \{v_1, \ldots, v_N\}$ of the input so decoding need not recompute them (Pope et al., 2023; Shazeer, 2019; Sheng et al., 2023); at step t the attention output is $o_t = \mathrm{Softmax}(q_t K^\top/\sqrt{d})\, V$ (Vaswani et al., 2017). Because the cache grows linearly in N, eviction is the problem of retaining $B < N$ entries while preserving what later generation needs: given an importance score s_i, a policy keeps $\mathcal{K}_B = \mathrm{TopB}(\{s_i\})$, and its effectiveness is how well s ranks the entries that will matter during generation.

To define an oracle target, assume the future response $Y = \{y_1, \ldots, y_T\}$ were available, and let $A_{\tau,i}$ be the attention assigned by response query τ to prompt token i. The *oracle future-attention importance* of token i is

$$s_i^{\mathrm{GT}} = \frac{1}{T} \sum_{\tau=1}^{T} A_{\tau,i}.$$

At inference Y is unknown, so s^{GT} must be approximated from what is available at the eviction point.

2.2 Two Families of Importance Signal

Existing policies estimate s_i in one of two ways. *Observed importance* reads behavior already visible at the eviction point: SnapKV (Li et al., 2024) and R-KV (Cai et al., 2025a) score prefix entries by attention over an observation window near the end of the prompt, H2O (Zhang et al., 2023) accumulates attention over the whole prompt, and LaProx (Mai & Kim, 2026) and LAVa (Shen et al., 2025) additionally account for the value representation and output projection. *Predicted future importance* instead estimates how important an entry will become in a response not yet generated — ForesightKV (Dong et al., 2026) and KVP (Moschella et al., 2026) train scorers on future-derived supervision, LookaheadKV (Ahn et al., 2026) approximates the response-induced distribution with learnable lookahead tokens and LoRA, and surrogate-generation methods use a draft response's attention as a proxy. Appendix F describes each policy.

What separates the families is the scope of evidence, not the scoring rule: the first is confined to dependencies visible before the eviction point, the second pays extra computation to guess the ones that are not.

3 MOTIVATION

3.1 COMPLEMENTARITY OF OBSERVED AND FUTURE IMPORTANCE

At the end of prefill we compare the top-B selections of the three signals: observed importance $\mathcal{K}_{\text{obs}}$, here SnapKV's own score; predicted future importance $\mathcal{K}_{\text{pred}}$, here the policy-agnostic full-future scorer of Section 5.4 trained on the oracle; and the oracle $\mathcal{K}_{\text{GT}}$ itself. Position is a coarse summary of how they differ, so we decompose the oracle set into the parts each uniquely recovers (Appendix G). Averaged over 293 documents drawn from 15 of the 16 LongBench tasks (all but PassageRetrieval-en) at $B = 128$, 23.0% of oracle-critical entries are recovered by both signals, 7.1% *only* by observed importance and 24.6% *only* by predicted future importance; the observed-only share ranges from 2.3% to 13.2% across tasks. A system already running an observed policy holds the 30.1% its own signal reaches and misses the rest, and that remainder is the rescue set $\mathcal{R} = \mathcal{K}_{\text{GT}} \setminus \mathcal{K}_{\text{base}}$ of Section 4. The 24.6% is the part of $\mathcal{R}$ a future signal already reaches: neither signal is a noisier copy of the other.

The split is not fixed across budgets (Figure 1). As the budget tightens, mass leaves *shared* for *missed by both*: aggressive compression is the regime in which a base policy leaves most on the table, and the observed-only share is smallest there (7.1% at $B = 128$), so almost all of what tightening costs falls outside what the observed signal alone can reach.

We state this as exclusive recall rather than as the union, which holds $1.64B$ entries and is not budget-matched. Matched properly, a predictor alone is the better way to spend a *fresh* cache on coverage (47.6% against 49.2% for a blend), though the same comparison run end to end reverses on all five bases (Appendix I). The premise here is a system already committed to an observed policy, and what survives budget matching is the asymmetry above: the 24.6% is what such a system discards.

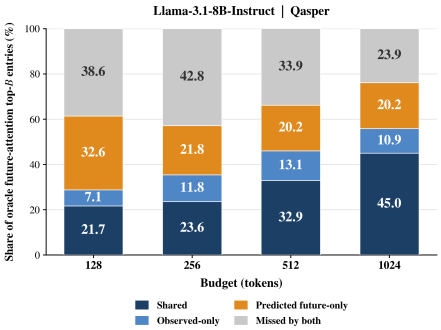


Figure 1: Which signal recovers each oracle-critical entry on Qasper (Dasigi et al., 2021), by budget. Here the observed signal is the mean percentile rank of the five base policies and the predicted one that of FORESIGHTKV and LOOKAHEADKV, so the shares differ from the 15-task averages quoted in the text, which use SnapKV against the full-future scorer.

3.2 OBSERVATION BOTTLENECK IN OBSERVED IMPORTANCE

If observed policies differed mainly in scoring rule, their mistakes would be largely independent and a better rule would reach what they miss. They are not. Over 15 tasks the four policies reading the same observation window rank tokens at pairwise Spearman 0.763–0.971, and all five miss the same oracle entries 1.03 to 1.37 times as often as independent failures would produce — highest for those four, lowest for every pair containing H2O, the one policy that accumulates attention over the whole prompt. What the policies can see tracks their agreement more closely than how they score it. We call this the *observation bottleneck*, naming a scope restriction rather than asserting a cause; the residual missed by only one policy (1.4–26.4%) is what makes the rescue set policy-specific (Appendix G).

4 RESCUE

RESCUE augments an existing eviction policy with two components: a policy-specific residual scorer that predicts future-critical KV entries missed by the base policy, and a fidelity-guided selector that determines whether the correction should be applied for each input. Figure 2 illustrates the overall pipeline.

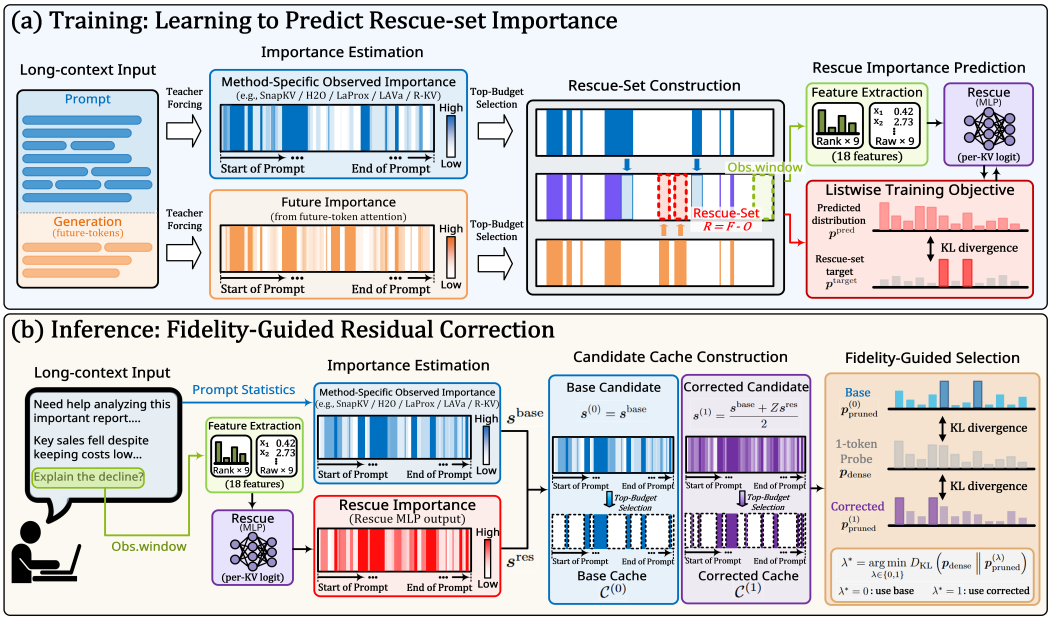


Figure 2: RESCUE. **Training**: the rescue set $\mathcal{R} = \mathcal{K}_{\mathrm{GT}} \setminus \mathcal{K}_{\mathrm{base}}$ is the part of the oracle future-attention selection the base policy drops, and the residual scorer is supervised on it alone. **Inference**: the base and corrected scores give two caches at the same budget, and one probe token decoded from the still-dense prompt cache picks the one whose output distribution is closer to the dense model's.

4.1 Learning Base-Specific Rescue Scores

From future prediction to residual prediction. Existing future-aware methods predict the future importance of every entry, which also reranks what the base policy already places correctly. RESCUE targets only the difference. Under budget B, let $\mathcal{K}_{\mathrm{base}} = \mathrm{TopB}(s^{\mathrm{base}})$ be what the base policy retains and $\mathcal{K}_{\mathrm{GT}} = \mathrm{TopB}(s^{\mathrm{GT}})$ the oracle set induced by attention over the realized continuation. The *rescue set* is

$$\mathcal{R} = \mathcal{K}_{\mathrm{GT}} \setminus \mathcal{K}_{\mathrm{base}},$$

and supervision is assigned to $\mathcal{R}$ alone; entries the base policy already keeps need not be relearned. We call s^{GT} an *oracle future-attention* score rather than ground truth because attention mass is a proxy for causal contribution, not a measurement of it (Section 5.5).

Scope of a selection. All sets are per layer over the candidate set $\mathcal{C}$ of evictable positions, so B counts *selectable* slots: the budget net of the 4 sink positions and the 32-token observation window, so $128 - 4 - 32 = 92$ at $B = 128$. This size defines the supervision target only; at inference each base policy's own protected window (Appendix B.1) governs what is retained. A (document, layer) pair with $\mathcal{R} = \emptyset$ has no defined target and is skipped. Because policies miss different entries, we share the features and objective but train one scorer per policy, and, since the target moves with B, one per budget as well (Appendix B.1).

That skip predicts where the correction fails. Trained only on pairs that *have* something to rescue, the scorer never learns to represent "nothing needs recovering" and applied unconditionally displaces entries even where the base was already right. The selector supplies that state externally, so it is not an add-on but the component representing the case the objective cannot express.

Features and listwise objective. Each candidate carries nine prefill-available features — five observation-window QK statistics (current, mean, maximum, variance, slope), $\|K_i\|$, $\|V_i\|$, $\|V_i W_O\|$ and relative age — concatenated with their population ranks, giving $x_i \in \mathbb{R}^{18}$; no extra forward pass is required. An MLP f_θ ($18 \to 32 \to 16 \to 1$) produces a logit z_i. Because eviction selects by relative rank under a fixed budget, training is listwise: with $p^{\mathrm{res}} = \mathrm{softmax}(z)$ over $\mathcal{C}$ and p^{target} uniform

on $\mathcal{R}$, $\mathcal{L}_{\text{res}} = D_{\text{KL}}(p^{\text{target}} \parallel p^{\text{res}})$, which learns the priority of the entries to recover rather than regressing attention magnitudes (Appendix B.2).

4.2 Fidelity-Guided Correction Selection

Because the correction is not beneficial for every input, RESCUE builds both candidate caches and selects between them by their fidelity to the dense model.

We take $s_i^{\text{res}} = p_i^{\text{res}}$, the softmax-normalized rescue distribution rather than the raw logit, so both scores are nonnegative and s^{res} sums to one over $\mathcal{C}$. The two live on unrelated scales, so the correction is put on the base's before mixing, by multiplying it by the base's total mass:

$$s_i^{(\lambda)} = \frac{s_i^{\text{base}} + \lambda\, Z\, s_i^{\text{res}}}{1+\lambda}, \qquad Z = \sum_j s_j^{\text{base}}, \qquad \lambda \in \{0, 1\},$$

so $\lambda = 0$ recovers the base policy and $\lambda = 1$ activates the correction, giving two candidate caches under the same budget.

Which candidate better preserves model behavior is measured, not predicted from prompt statistics. The dense cache is still available immediately after prefill, so RESCUE greedily generates one probe token from it and records the full-vocabulary distribution p_{dense}. Evaluating the same token under the two candidates gives $p^{(0)}_{\text{pruned}}$ and $p^{(1)}_{\text{pruned}}$, and

$$\lambda^* = \arg\min_{\lambda \in \{0,1\}} D_{\text{KL}}\left(p_{\text{dense}} \parallel p^{(\lambda)}_{\text{pruned}}\right),$$

with ties resolved conservatively to $\lambda^* = 0$; the check is empirical, not a guarantee. The probe and its KV state are then discarded and generation restarts from the selected cache, so the probe never enters the final context. Longer probes are evaluated in Appendix O.1.

Abstaining when the probe carries no evidence. In float32 a divergence summed over the vocabulary carries an absolute error near 10^{-7}, and where both candidates already track the dense distribution the true value falls below it, leaving the $\arg\min$ to rank rounding noise — 61.5% of PassageRetrieval documents on Qwen3-8B against 0% on Llama-3.1. RESCUE therefore abstains: if every candidate falls below $\tau = 10^{-6}$ it keeps the base cache, since no evidence is not evidence for the correction. On Qwen3-8B with SnapKV that is the difference between -0.16 and $+0.47$ (Appendix N). The probe is thus a test-time verification budget — one dense decoding step and the dense prompt cache it requires — which Section 5.4 quantifies.

5 Experiments

5.1 Experimental Setup

We evaluate on Meta-Llama-3.1-8B-Instruct, Mistral-7B-Instruct-v0.3 (Jiang et al., 2023) and Qwen3-8B (Yang et al., 2025) across all 16 LongBench tasks with the official metrics, augmenting SnapKV, LaProx, LAVa, R-KV and H2O — the last in this paper's evict-once form, not its published decode-time schedule — and comparing against ForesightKV and LookaheadKV. All methods retain 128 prompt KV positions *per layer*, written $B_{\text{total}} = 128L$; each base policy keeps its own protected recency window, which RESCUE inherits unchanged. Scorers are trained on Natural Questions and arXiv documents disjoint from LongBench, and checkpoints, prompt templates and decoding configuration are identical across methods. Every analysis beyond the main table is on Llama alone (Appendix B.1).

5.2 Main Results

Table 1 summarizes LongBench over the complete grid of 80 task–base cells on Llama-3.1-8B-Instruct. RESCUE gains $+1.28$ points on average (95% CI $[+0.51, +2.16]$ over 10,000 task-level resamples), with a median of $+0.52$ and a worst cell of -2.17; at a 0.05-point margin it wins 51, loses 24 and ties 5, and improves every base policy. H2O, the weakest base, gains most ($+2.51$) and carries 39% of the mean: the other four average $+0.97$, and $+0.84$ on Mistral, $+0.12$ on Qwen3.

Table 1: LongBench at $B_{\text{total}} = 128L$, averaged over the 16 tasks. Each base policy is paired with the same policy under RESCUE, because the rescue set is defined by that policy's own misses and a shared reference would mean nothing. Intervals are hierarchical bootstraps resampling *tasks*, since the 80 cells reuse the same 16 tasks. Per-task scores are in Table 3 (Appendix A).

	Llama-3.1-8B		**Mistral-7B-v0.3**		**Qwen3-8B**	
Base policy	Base	+ RESCUE	Base	+ RESCUE	Base	+ RESCUE
Dense (no eviction)	*49.95*		*47.69*		*49.43*	
LaProx	43.88	**44.70**	41.28	**41.85**	**43.17**	42.94
H2O (evict-once)	43.04	**45.55**	38.51	**40.57**	39.60	**41.22**
SnapKV	43.03	**44.74**	39.60	**41.16**	42.12	**42.59**
LAVa	43.88	**44.97**	41.30	**42.25**	**43.20**	43.11
R-KV	45.03	**45.30**	41.87	**42.15**	40.97	**41.31**
Mean Δ	$+\mathbf{1.28}$		$+\mathbf{1.08}$		$+\mathbf{0.43}$	
95% CI	$[+0.51, +2.16]$		$[+0.15, +2.26]$		$[+0.13, +0.87]$	
Mean Δ, excluding H2O	$+0.97$		$+0.84$		$+0.12$	

Compressing to 128 positions per layer costs 6.18 points, leaving the base policies at 87.6% of the uncompressed score; RESCUE raises that to 90.2%, recovering 20.7% of what eviction costs. The share varies by task, from 83% on NarrativeQA to 1% on GovReport and MultiNews.

The same grid reproduces the effect on two further backbones without retuning: Mistral-7B-Instruct-v0.3 gains $+1.08$ $[+0.15, +2.26]$ (48 wins, 25 losses) and Qwen3-8B $+0.43$ $[+0.13, +0.87]$ (42 against 28), both intervals excluding zero. What transfers is not only the sign but the ordering — H2O gains most, R-KV least — which is the residual claim arriving by a second route: the base that leaves the most unrecovered is the base a correction helps most. So do the failure modes, the two code tasks losing on nine of their ten Mistral pairs, and on Qwen3 two policies land just under zero (LaProx -0.23, LAVa -0.09) at about a third the size seen elsewhere. Across all three the sign and the ordering transfer; the magnitude does not.

LongBench prompts average 5K tokens, so the same budget compresses far harder at longer input. On RULER's five retrieval tasks RESCUE gains $+5.72$, $+2.92$ and $+2.12$ points at 16K, 32K and 64K, but that tracks how far the base has fallen rather than how long the input is: H2O, which keeps under a tenth of the dense score at every length, carries it, while the four policies that stay functional gain $+0.05$, $+0.45$ and $+0.50$, rising as they themselves lose ground (Appendix J).

Among dedicated future-aware methods the informative comparator is LookaheadKV, the one that stays competitive at this budget: pooled over the five augmented policies (paired against a shared reference, so not independent) RESCUE is $+1.68$ ahead, paying 277 ms of added logic against its 11.2 ms (Appendix L). The margin is accuracy bought with latency, and the claim is competitiveness rather than dominance — LookaheadKV matches or exceeds RESCUE on several tasks and collapses on TREC (Appendix A.1). ForesightKV cannot run at its published recency window under this budget, and at the widest window leaving its scorer any slots it scores 40.43, below every base policy here, so the $+4.62$ margin over it is a property of the operating point rather than of the correction (Appendix B.1).

5.3 Analysis of RESCUE

Does the correction recover what the base policy missed? We measure what each selection holds, not only how the scores move. At $\lambda = 1$ over the 74 cells with both measurements, RESCUE raises oracle coverage by 5.0 points (30.8% $\rightarrow$ 35.8%), recovers 25.7% of the rescue set, and $\Delta\text{Coverage}$ tracks the downstream gain ($\rho = 0.476$). The score changes are therefore associated with retaining entries the base policy dropped, not with reshuffling its ranking (Appendix H).

The same holds one level down. Over all 18,750 documents the unconditional correction is a coin flip in the tail — it loses at least 5 points on 14.5% and wins as much on 14.1% — while gating halves the losing share to 6.0%, and the win-to-loss ratio rises the further out one looks, from 1.22 at a point to 1.87 at twenty (Appendix N.3). The selector is worth most where a mistake costs most.

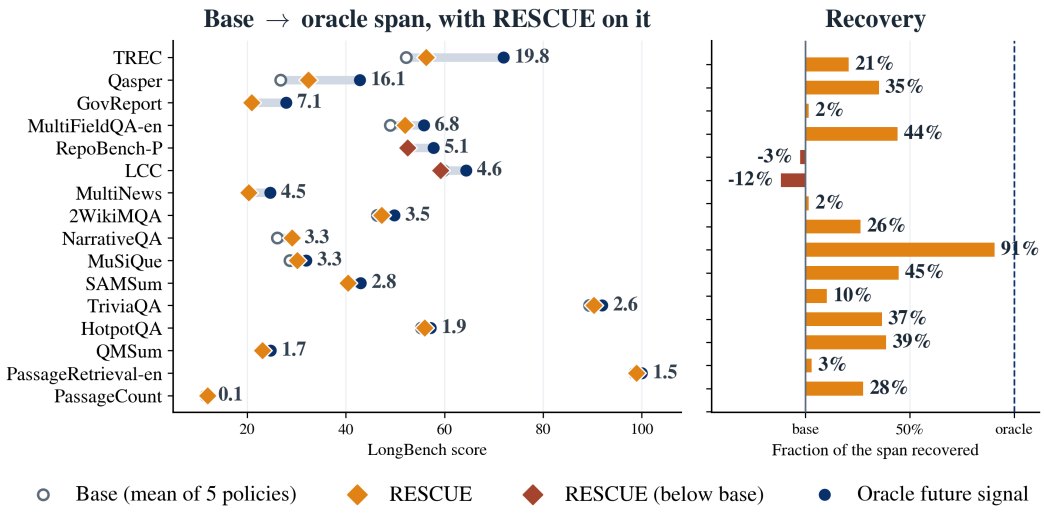


Figure 3: How far RESCUE moves from each base policy towards an oracle future signal at a 128-token budget, over 16 tasks. **Left**: the base-to-oracle span with RESCUE on it, ordered by span size. **Right**: the same as a fraction; negative means the correction ends farther from the oracle than the base (Appendix K).

Where the correction fails, and why the selector matters. The per-policy breakdown is the sharpest result here, because the sign flips (Table 9, Appendix H). On H2O, the weakest base, the correction recovers 46.3% of what the policy missed and raises coverage by 25.7 points; on R-KV, the strongest, it recovers 0.7% and *lowers* coverage by 20.8, displacing entries R-KV had already selected correctly at a cost of 0.70 points and a worst cell of -8.50. R-KV still ends at $+0.27$ under the full method, because the selector declines the correction on most of its documents.

By task the two do not overlap: on TREC the correction alone is -1.60 and the selector turns it into $+4.10$, while on summarization the correction alone is near zero and there is nothing to accept (Appendix M).

How much of the achievable gain remains unclaimed. Replacing the learned scorer with the continuation's true future attention bounds what is available to win (Figure 3). A task's *headroom* — that oracle score minus the mean of the five base policies — averages 5.30 points, of which RESCUE recovers 24.4%, and it predicts the gain only weakly: four tasks reach under a tenth of more than three points available (Appendix K).

5.4 Ablations and Overhead

Probe length, the number of candidate λ and a cap on contested slots are each set against their cost rather than for accuracy: we ship probe length 1, two candidates and no cap, forgoing the $+0.43$ points longer probes buy at $+54$ ms each (Appendix D).

Correction and fidelity selection. Table 2 isolates the two contributions under both learning targets.

Applied unconditionally the residual correction gains $+0.64$ $[-0.42, +1.89]$, an interval containing zero, and does so unsafely: 18 of 80 cells lose at least a point and the worst loses 9.50 (Figure 4). Gating it with the fidelity selector doubles the gain to $+1.28$ $[+0.51, +2.16]$ and cuts the harmful cells to 3. The full-future target repeats the pattern, $+0.47$ $[-0.29, +1.25]$ against $+1.06$ $[+0.43, +1.77]$: **whichever target is used, the arm that excludes zero is the gated one.** Per base policy the gain excludes zero on three of five (Appendix M).

Against one full-future scorer serving all five policies the residual target is ahead in both regimes ($+0.17$ and $+0.22$) without either interval excluding zero; where the two separate is the tail (Ap-

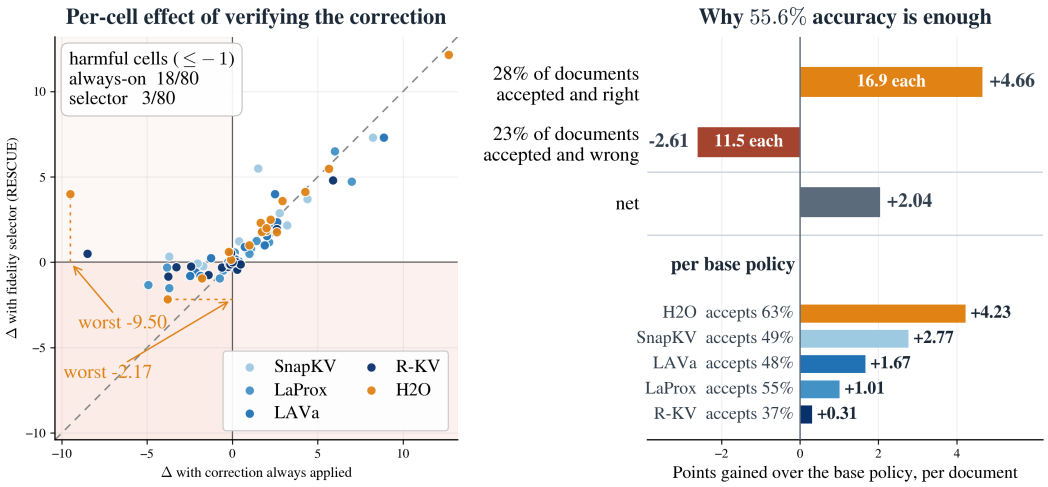


Figure 4: Verification at two levels. **Left**: each (base, task) cell with the correction always applied (x) against the same correction gated by the selector (y); the dotted drops mark the two extremes, both H2O. **Right**: the same over the 9,252 documents where the two caches differ, where the decisions the selector gets right are larger than the ones it gets wrong (Appendix N.3).

Table 2: Ablation over all 80 (base, task) cells, paired on the same cells. The two targets differ only in whether the base policy's correct selections are excluded from supervision. "Harmful" counts cells losing at least a point. The oracle row takes the better cache per document in hindsight, bounding any selection rule of this form (Appendix P.1).

Method	**Avg. score**	**Δ vs. Base**	**Worst Δ**	**Harmful cells**
Base	43.77	–	–	–
Correction applied unconditionally:				
Full-future target	44.24	+0.47	−9.00	17 / 80
Residual target	44.41	+0.64	−9.50	18 / 80
Correction gated by the fidelity selector:				
Full-future target	44.84	+1.06	−1.93	7 / 80
Residual target (RESCUE)	**45.05**	**+1.28**	−2.17	**3 / 80**
Oracle per-input selector	47.32	+3.54	–	–

pendix P.1). The split across base policies follows the coverage measurements of Section 5.3: the selector adds +0.97 on R-KV, where unconditional correction is harmful, and +1.05 on H2O, where it already helps; per cell it wins 38, loses 32 and ties 10.

Is the scorer necessary if the selector works? The selector is the cheaper component, so two controls bound what it does alone. Asked to choose between SnapKV's and ForesightKV's *existing* caches rather than two values of λ, it stops well short: +0.50 against the residual scorer's +1.71 on the same base (Appendix N.1). A selector is bounded by the candidates it is handed.

The cheaper control is the stronger one. The probe token's attention over the prompt is itself a sample of the future, so adding that row to SnapKV's own vote — no scorer, no second cache, no divergence — costs nothing beyond the probe already decoded and gains +1.95 on the 16-task mean against RESCUE's +1.71. That mean is one task: on TREC the first generated token lands on the in-context labels the prompt is built from and the row alone gains +20.50 against +5.50, while over the other fifteen tasks it gains +0.71 against +1.46, is behind on nine, and its worst task costs −0.86 against −0.22. Where the first generated token already attends to what matters, one attention row suffices; where it does not, the learned residual is what recovers the entries. Combining them — the probe row as a tenth feature — is untried. Appendix N.1 reports the control in full, including its weight sweep.

Does the budget matter? The residual view predicts the correction is worth more the more the base policy must drop. Repeating the grid at $B = 256$ and $B = 1024$ tests that: RESCUE gains $+1.28$, $+0.88$ and $+0.28$, and both steps are separated by the task bootstrap ($+0.40$ $[+0.07, +0.78]$ and $+0.61$ $[+0.16, +1.12]$), though the first does not survive the seed correction of Appendix P. At $B = 1024$ the correction is worth nothing on the four strongest base policies, which already retain most of the oracle set, while H2O, the weakest, still gains $+1.48$. The base policies gain far more from the extra budget than RESCUE does ($43.77 \rightarrow 48.69$): the room a correction works in closes as the budget opens (Appendix A.2).

Runtime and memory overhead. At an average prompt of 5K tokens, eviction- and selection-related latency is 5.6 ms for SnapKV and 134.7 ms for ForesightKV against 277.2 ms for the complete one-token pipeline: about twice the most expensive predicted-future method we compare against, so the margin in Section 5.2 is accuracy bought with latency. End to end it is 49% of time-to-first-token against SnapKV's 1.9%, almost none of it the scorer: the probe costs 21 ms against 254 ms to build and replay the candidates. Peak memory is 23.0 GB against 17.1 GB, but that excess is a prefill-time spike rather than a standing cost: RESCUE decodes with its base policy's cache, not a larger one (Appendix L). It does assume an evict-once setting and does not support streaming or chunked prefill.

Against simply raising the budget. The comparison a deployment faces is not RESCUE against another eviction rule at the same budget but against a larger cache. Where the budget is free to raise, raising it is the better trade: SnapKV at $B = 256$ scores $+0.96$ above RESCUE at $B = 128$ and pays 272 ms less per document. The two costs are not paid in the same way, though: a larger cache is paid per token for the whole of decoding and per sequence in a batch, while RESCUE's is paid once after prefill, so the ordering reverses once generation is long enough. Measured against decode throughput, it reverses at about 2,000 generated tokens for any batch of four or more, and not until 11,600 at batch one (Appendix D.1). What bounds the regime instead is that RESCUE's overhead is flat in the budget (277.2, 275.1, 274.2 ms, since the cost scales with the prompt) while the gain is not ($+1.71$, $+0.98$, -0.02).

Concretely, at high batch the budget is paid in throughput and not memory alone: at batch 128, $B = 256$ decodes 30% slower than $B = 128$ and $B = 1024$ nearly four times slower, while the uncompressed cache no longer fits (Table 8). A deployment holding a small budget for that reason, and generating past the break-even, is the one this method is for: there the alternative to $+1.71$ points is not a larger cache but nothing.

5.5 Scope of the Evidence

Three claims rest on narrower evidence than the main grid. The residual target's advantage over a policy-agnostic one is resolved on two base policies and three tasks, not the full grid, where the margin is $+0.22$. The selector earns its average from the size of the decisions it gets right rather than their number, agreeing with hindsight on 55.6% of deciding documents. And s^{GT} is future attention mass rather than measured causal contribution, so the target is a proxy for the one that matters (Appendix E).

Two controls remain unrun and are what we run next: deferring the base policy's own eviction by the decode steps the probe costs, the cheapest competing use of the same held memory, and transferring a scorer across base policies, which would separate what the residual target learns about a policy from what it learns about the task.

6 Conclusion

Observed and predicted future importance recover complementary subsets of future-critical entries, and policies reading the same evidence fail on largely the same ones: the limit is what is visible at eviction, not the scoring rule. **RESCUE** therefore corrects a deployed policy rather than replacing it, and verifies the correction first — applied unconditionally it harms 18 of 80 cells; gated on a single-token probe it doubles the gain and leaves 3.

Reproducibility Statement

Code, trained scorers and every number in this paper are at `https://anonymous.4open.science/r/RESCUE-54E0`.

The evaluation protocol is specified in Appendix B.1: budgets, protected windows, per-model context limits, the five LongBench tasks that never receive a chat template, and the deviations from each baseline's published settings. Training is specified in Appendix B.2.

The repository is arranged so that each claim can be checked at a different cost. Every table and figure is generated by a script named for its number in this paper, reading two committed files — all 1,324 benchmark cells and all 18,750 per-document scores — so the numbers here can be reproduced with no GPU, no model weights and no benchmark data. Underneath those, raw model generations for one cell per LongBench metric family are included with the references they were scored against, and a script recomputes their scores with the official metric, so the aggregates rest on text a reader can read rather than on numbers in a file. Regenerating those files from evaluation output, and the evaluation output itself, are separate entry points. The 26 trained scorers are included, so the evaluation can be run without repeating training.

AI Assistance Disclosure

In this work, we used generative AI tools to assist with research ideation, retrieval and discovery of related work, drafting portions of the manuscript, and editing and polishing the writing. The authors reviewed and verified all AI-assisted content, including technical claims, references, experimental descriptions, and reported results. Generative AI tools were not used to generate synthetic datasets or to prove mathematical claims. The authors take full responsibility for the final content of this work.

References

Jinwoo Ahn, Ingyu Seong, Akhil Kedia, Junhan Kim, Hyemi Jang, Kangwook Lee, and Yongkweon Jeon. LookaheadKV: Fast and accurate KV cache eviction by glimpsing into the future without generation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=RVLMGPXt2i`.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 3119–3137, 2024.

Payman Behnam, Yaosheng Fu, Ritchie Zhao, Po-An Tsai, Zhiding Yu, and Alexey Tumanov. RocketKV: Accelerating long-context LLM inference via two-stage KV cache compression. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 3358–3392. PMLR, 13–19 Jul 2025. URL `https://proceedings.mlr.press/v267/behnam25a.html`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Zefan Cai, Wen Xiao, Hanshi Sun, Yikai Zhang, Ke Wan, Yucheng Li, Yeyang Zhou, Li-Wen Chang, Jiuxiang Gu, Zhen Dong, et al. R-kv: Redundancy-aware kv cache compression for reasoning models. *Advances in neural information processing systems*, 38:60980–61005, 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*, 2025b. URL `https://openreview.net/forum?id=ayi7qezU87`.

Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 2 (short papers)*, pp. 615–621, 2018.

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

Zican Dong, Peiyu Liu, Junyi Li, Zhipeng Chen, Han Peng, Shuo Wang, and Wayne Xin Zhao. Foresightkv: Optimizing kv cache eviction for reasoning models by learning long-term contribution. *arXiv preprint arXiv:2602.03203*, 2026.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *Advances in Neural Information Processing Systems*, 38:113152–113188, 2025.

Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pp. 70–79, 2019.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*, pp. 12098–12107. PMLR, 2023.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, 2017.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026/.

Xin Li and Dan Roth. Learning question classifiers. In *Coling 2002: The 19th international conference on computational linguistics*, 2002.

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

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, 2024.

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

Tho Mai and Joo-Young Kim. Reformulating KV cache eviction problem for long-context LLM inference. *arXiv preprint arXiv:2605.07234*, 2026. URL https://arxiv.org/abs/2605.07234.

Luca Moschella, Laura Manduchi, and Ozan Sener. Learning to evict from key-value cache. *arXiv preprint arXiv:2602.10238*, 2026.

Piotr Nawrot, Adrian Łańcucki, Marcin Chochowski, David Tarjan, and Edoardo M. Ponti. Dynamic memory compression: Retrofitting LLMs for accelerated inference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 37396–37412. PMLR, 2024.

Matanel Oren, Michael Hassid, Nir Yarden, Yossi Adi, and Roy Schwartz. Transformers are multi-state rnns. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 18724–18741, 2024.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of machine learning and systems*, 5:606–624, 2023.

Ziran Qin, Yuchen Cao, Mingbao Lin, Wen Hu, Shixuan Fan, Ke Cheng, Weiyao Lin, and Jianguo Li. Cake: Cascading and adaptive kv cache eviction with layer preferences. In *International Conference on Learning Representations*, 2025.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Yiqun Shen, Song Yuan, Zhengze Zhang, Xiaoliang Wang, Daxin Jiang, and Cam-Tu Nguyen. LAVa: Layer-wise KV cache eviction with dynamic budget allocation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 13672–13692, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.737. URL https://aclanthology.org/2025.findings-emnlp.737/.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/sheng23a.html.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. QUEST: Query-aware sparsity for efficient long-context LLM inference. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 47901–47911. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/tang24l.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Guangtao Wang, Shubhangi Upasani, Chen Wu, Darshan Gandhi, Jonathan Li, Changran Hu, Bo Li, and Urmish Thakker. LLMs know what to drop: Self-attention guided KV cache eviction for efficient long-context inference. *arXiv preprint arXiv:2503.08879*, 2025.

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

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

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

A PER-TASK AND PER-POLICY TABLES

Table 3 gives the per-task scores behind the summary of Table 1, for all three backbones.

A.1 PER-POLICY COMPARISON WITH FUTURE-AWARE METHODS

Table 4 breaks the pooled comparison of Section 5.2 down per base policy. In absolute terms the two comparators average 43.38 (LookaheadKV) and 40.43 (ForesightKV) over the 16 tasks on Llama-3.1-8B-Instruct, against 44.70 to 45.55 for the five RESCUE arms of Table 1. Cells where a margin is below 0.05 points count as neither a win nor a loss, so a row need not sum to 16.

A.2 BUDGET SWEEP, PER TASK

Section 5.4 reports the budget sweep as five averages per budget. The same 80 cells are given per task at $B = 256$ and $B = 1024$ in Table 5, in the layout of Table 1 so the three budgets can be read against one another; the $B = 128$ block is Table 1 itself. Bold marks the better of the two rows in each pair, as there.

A.3 RULER, PER TASK

Table 6 gives the per-task scores behind Table 11, at each context length.

One column is zero everywhere and is worth explaining. The answer `niah_multikey_3` asks for is a 32-character random UUID, and the metric scores an inexact string zero. The dense model returns it on 99 of 100 documents; at a 128-token budget no policy returns it even once, with or without the correction — what survives eviction is enough to begin the string and not enough to finish it. The column is therefore a property of the budget rather than of any arm: it lowers all eleven rows by the same amount and does not move any comparison between them.

Table 3: Per-task scores behind Table 1: all 16 LongBench datasets, on all three backbones. Bold marks the better of each base/+RESCUE pair, per column. †The average is over the 16 tasks, unweighted by document count.

Method	Single Document QA			Multi Document QA			Summarization			Few-shot Learning			Synthetic		Code		Avg.†
	NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	MuSiQue	GovRep	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PR-en	Lcc	RB-P	
Meta-Llama-3.1-8B-Instruct, $B_{\text{total}} = 128L$																	
Dense (no eviction)	29.74	44.96	55.61	57.61	48.69	33.76	34.70	25.40	26.88	72.50	91.66	43.92	10.10	100.00	64.79	58.89	49.95
LaProx	26.98	27.83	**53.41**	**56.05**	**47.91**	29.17	19.11	22.86	18.82	47.50	88.28	**40.66**	**12.00**	98.00	**59.58**	**53.96**	43.88
+ RESCUE	**29.23**	**32.55**	52.92	55.77	46.96	**29.97**	**20.37**	**22.97**	**19.39**	**54.00**	**89.46**	40.35	**12.00**	**98.50**	58.08	52.63	**44.70**
H2O (evict-once)	25.12	26.38	37.57	53.46	44.91	27.55	**24.43**	22.97	**23.30**	59.50	88.01	38.40	10.00	98.50	58.99	49.58	43.04
+ RESCUE	**30.59**	**30.51**	**49.72**	**55.30**	**46.69**	**31.15**	22.26	**23.12**	22.35	**63.50**	**90.50**	**40.72**	**12.00**	**99.50**	**59.60**	**51.35**	**45.55**
SnapKV	24.40	25.15	49.44	55.98	46.22	28.08	18.68	22.93	18.86	47.00	89.33	40.23	**12.00**	98.00	**59.34**	**52.83**	43.03
+ RESCUE	**28.10**	**32.46**	**52.32**	**57.21**	**47.07**	**30.25**	**20.36**	**23.00**	**19.61**	**52.50**	**90.27**	**40.56**	11.80	**98.50**	59.12	52.76	**44.74**
LAVa	27.22	26.06	51.31	55.99	45.22	28.18	20.41	23.50	19.82	46.50	90.17	**40.68**	11.83	**99.25**	**60.87**	**55.05**	43.88
+ RESCUE	**29.03**	**33.37**	**52.30**	**56.89**	**47.57**	**29.72**	**20.47**	**23.54**	**20.11**	**50.50**	**90.41**	40.08	**12.00**	**99.25**	60.07	54.26	**44.97**
R-KV	26.69	28.45	**53.12**	**54.87**	47.57	**30.29**	**21.38**	**22.90**	**20.23**	60.50	**91.10**	**40.74**	**12.00**	**99.00**	**59.89**	**51.80**	45.03
+ RESCUE	**28.63**	**33.25**	52.69	**54.87**	**48.10**	29.55	21.12	22.78	19.92	**61.00**	90.98	40.46	11.88	**99.00**	59.05	51.56	**45.30**
Mistral-7B-Instruct-v0.3, $B_{\text{total}} = 128L$																	
Dense (no eviction)	26.95	38.47	50.62	50.59	38.92	28.75	34.25	25.78	26.59	76.00	88.89	47.79	7.00	98.00	61.60	62.84	47.69
LaProx	**24.74**	24.05	**47.23**	**48.97**	**34.83**	**25.20**	20.99	22.58	19.96	53.00	89.08	43.35	**5.50**	89.50	**56.66**	**54.83**	41.28
+ RESCUE	24.44	**27.08**	46.68	47.87	34.49	24.29	**21.48**	**22.72**	**20.13**	**66.00**	**89.19**	**43.38**	4.50	89.50	54.42	53.42	**41.85**
H2O (evict-once)	21.84	25.35	34.84	43.99	31.75	22.51	**25.42**	21.64	**23.14**	66.00	84.97	39.04	4.00	**64.50**	**54.93**	52.16	38.51
+ RESCUE	**25.29**	**29.97**	**44.69**	**47.15**	**36.06**	**23.83**	22.67	**22.16**	22.28	**72.50**	**88.77**	**41.89**	**6.00**	59.00	54.47	**52.33**	**40.57**
SnapKV	21.73	23.70	42.69	46.09	32.63	21.08	20.95	21.82	19.61	44.50	89.53	**42.59**	5.00	**91.50**	**55.60**	**54.60**	39.60
+ RESCUE	**24.09**	**30.00**	**44.11**	**46.81**	**35.88**	**23.28**	**21.80**	**22.65**	**20.01**	**56.00**	**89.54**	41.39	5.00	90.00	54.10	53.91	**41.16**
LAVa	23.62	26.29	46.41	47.14	34.28	24.39	21.98	22.50	20.49	52.50	**89.04**	**43.84**	4.00	90.50	**57.14**	**56.65**	41.30
+ RESCUE	**24.99**	**29.43**	**47.71**	**47.64**	**36.20**	**25.23**	**22.30**	**22.78**	**20.84**	**60.00**	88.48	43.07	**4.50**	90.50	56.46	55.85	**42.25**
R-KV	23.91	28.97	47.30	46.36	33.74	24.47	**22.52**	**22.78**	20.64	67.00	88.94	**42.86**	**4.50**	86.50	**55.77**	**53.70**	41.87
+ RESCUE	**24.29**	**29.47**	**48.08**	**48.78**	**33.85**	**25.61**	22.51	22.42	**20.70**	**68.50**	**89.45**	41.77	4.00	86.50	55.41	53.10	**42.15**
Qwen3-8B, $B_{\text{total}} = 128L$																	
Dense (no eviction)	26.36	48.26	53.87	58.93	43.29	35.26	33.60	23.99	24.83	71.50	90.71	44.01	2.00	100.00	69.05	65.26	49.43
LaProx	19.23	**32.58**	**51.66**	**55.24**	**40.46**	**28.84**	17.25	20.89	16.12	**56.00**	**89.70**	**39.70**	2.50	**97.00**	**64.69**	**58.89**	**43.17**
+ RESCUE	**19.33**	32.57	51.53	54.94	40.04	27.25	**19.58**	20.89	**16.41**	55.50	89.26	39.27	2.50	96.00	63.92	58.11	42.94
H2O (evict-once)	**18.11**	**32.54**	35.56	50.49	36.55	27.02	**25.86**	20.98	**21.09**	59.00	83.49	37.39	2.00	64.00	63.43	56.04	39.60
+ RESCUE	17.83	32.14	**38.68**	**52.40**	**37.36**	**28.07**	20.80	**21.04**	19.23	**60.50**	**85.43**	**39.79**	**2.50**	**79.83**	**65.50**	**58.49**	**41.22**
SnapKV	18.79	31.13	46.25	**55.30**	40.38	26.03	17.36	20.43	16.05	52.00	**88.93**	**38.71**	3.00	97.00	**64.41**	**58.12**	42.12
+ RESCUE	**20.08**	**32.41**	**46.87**	55.25	**40.84**	**27.19**	**18.50**	**20.77**	**16.45**	**52.50**	88.76	38.67	**4.00**	97.00	64.15	58.02	**42.59**
LAVa	**19.98**	**32.80**	47.98	56.42	39.86	**27.42**	18.77	21.08	**16.59**	53.00	88.75	39.09	**5.00**	99.00	**65.96**	**59.55**	**43.20**
+ RESCUE	19.24	32.09	**48.04**	**56.65**	**40.16**	27.10	**18.94**	**21.13**	16.53	**53.50**	88.75	**39.48**	4.50	99.00	65.54	59.18	43.11
R-KV	19.96	33.51	44.09	**55.14**	39.17	28.58	**20.28**	20.65	17.25	66.00	88.33	**39.57**	**4.00**	59.50	62.95	56.60	40.97
+ RESCUE	**20.43**	**34.18**	**44.54**	54.60	39.17	**29.23**	20.19	**20.74**	**17.30**	**67.00**	88.33	39.23	3.50	**61.50**	**63.53**	**57.57**	**41.31**

Table 4: RESCUE against future-aware comparators, per base policy, on the 16 LongBench tasks. Cells are counted as wins/losses at a 0.05-point margin. ForesightKV is run with its recency window reduced from its published 128 to 64; at the published value that window consumes the entire per-layer budget and its scorer receives no slots at all (Qasper 3.55, degenerate output), so the comparison would otherwise be vacuous.

	vs. LookaheadKV		vs. ForesightKV	
Base + RESCUE	**W/L**	**Mean Δ**	**W/L**	**Mean Δ**
H2O	12 / 4	+2.20	14 / 1	+5.15
LAVa	10 / 6	+1.61	13 / 2	+4.56
SnapKV	9 / 7	+1.36	13 / 3	+4.31
LaProx	9 / 7	+1.32	13 / 2	+4.27
R-KV	8 / 7	+1.93	13 / 3	+4.88

Table 5: LongBench at $B_{\text{total}} = 256L$ and $1024L$, Llama-3.1-8B-Instruct. †The average is over the 16 tasks, unweighted by document count.

Method	Single Document QA			Multi Document QA			Summarization			Few-shot Learning			Synthetic		Code		Avg.†
	NrtvQA	Qasper	MF-en	HotpotQA	2WikiMQA	MuSiQue	GovRep	QMSum	MultiNews	TREC	TriviaQA	SAMSum	PCount	PR-en	Lcc	RB-P	
$B_{\text{total}} = 256L$																	
SnapKV	26.71	31.76	**54.03**	**57.38**	47.76	29.53	21.58	**24.22**	21.71	55.00	**91.58**	41.08	11.83	99.50	**61.90**	**55.71**	45.70
+ RESCUE	**29.64**	**36.83**	53.44	57.16	**49.73**	**31.18**	**22.55**	24.08	**22.45**	**59.00**	91.53	**41.11**	**12.08**	99.50	61.69	54.96	**46.68**
LaProx	29.12	34.06	**54.57**	**56.74**	48.75	**32.18**	22.41	**24.11**	21.68	58.00	90.39	41.43	11.80	99.50	**63.54**	**56.51**	46.55
+ RESCUE	**30.49**	**36.52**	54.05	56.73	**49.10**	31.27	**22.82**	23.91	**21.71**	**60.50**	**90.96**	**41.78**	11.80	99.50	62.58	56.27	**46.87**
LAVa	25.87	34.63	**55.42**	**56.82**	46.78	30.30	23.11	23.59	21.99	62.50	91.61	41.74	**11.75**	100.00	62.93	**56.71**	46.61
+ RESCUE	**29.38**	**37.77**	55.09	56.65	**48.24**	**31.43**	**23.23**	**24.52**	**22.37**	**64.00**	**91.86**	**42.09**	11.71	100.00	**63.07**	56.07	**47.34**
R-KV	28.94	34.75	54.54	56.15	47.39	31.27	**23.66**	23.48	**22.79**	65.50	91.53	**41.88**	**11.75**	100.00	**61.34**	**54.43**	46.84
+ RESCUE	**29.47**	**36.33**	**55.20**	**56.30**	**47.58**	**31.59**	23.55	**23.62**	22.69	65.50	91.53	41.80	11.67	100.00	61.31	54.14	**47.02**
H2O (evict-once)	26.72	31.79	44.23	54.87	45.93	28.52	**27.05**	23.29	**24.61**	64.00	89.23	39.03	10.00	96.00	61.29	51.99	44.91
+ RESCUE	**29.94**	**36.70**	**54.60**	**55.71**	**46.63**	**31.67**	24.15	**23.57**	24.03	**67.50**	**91.63**	**40.94**	**11.92**	**99.50**	**61.92**	**53.23**	**47.10**
$B_{\text{total}} = 1024L$																	
SnapKV	**30.74**	42.53	**56.53**	**57.61**	**48.76**	**32.13**	**27.15**	24.28	25.36	69.00	91.77	**42.13**	11.71	100.00	63.62	58.16	**48.84**
+ RESCUE	30.16	**42.58**	56.05	57.41	48.62	32.06	27.10	**24.85**	25.36	**69.50**	91.77	42.12	11.71	100.00	**63.74**	**58.17**	48.83
LaProx	29.52	**43.42**	**56.55**	57.33	48.43	32.40	**27.36**	24.58	**25.60**	68.00	**91.31**	**42.75**	11.79	99.50	64.59	58.00	48.82
+ RESCUE	**29.83**	43.16	56.39	**57.69**	**48.59**	**32.49**	27.20	**24.75**	25.57	**69.50**	91.04	41.91	11.79	99.50	**64.68**	**58.05**	**48.88**
LAVa	30.13	**43.75**	55.26	**57.87**	48.31	32.07	27.55	24.71	**25.57**	72.00	**91.89**	42.44	11.58	100.00	**64.33**	**58.77**	**49.14**
+ RESCUE	**30.20**	43.61	**55.31**	57.24	**48.35**	**32.17**	**27.65**	**24.85**	25.56	**72.50**	91.49	**42.65**	11.58	100.00	64.07	58.27	49.09
R-KV	**29.48**	42.74	**56.40**	**57.17**	**48.13**	**32.62**	**28.50**	24.69	25.70	71.50	91.72	**43.18**	11.62	99.50	63.59	57.31	**48.99**
+ RESCUE	29.13	42.74	55.95	57.13	47.72	32.40	28.46	**24.77**	25.70	71.50	91.72	42.88	11.62	99.50	**63.61**	**57.41**	48.89
H2O (evict-once)	28.28	40.22	52.92	56.84	**48.09**	31.25	**30.43**	24.57	**26.56**	69.00	90.04	41.59	9.83	93.00	63.58	56.14	47.65
+ RESCUE	**29.81**	**42.30**	**56.40**	**56.98**	47.73	**32.44**	30.33	**24.94**	26.48	**71.00**	**91.14**	**42.32**	**11.58**	**100.00**	**64.31**	**58.25**	**49.13**

Table 6: RULER per task at $B_{\text{total}} = 128L$ on Llama-3.1-8B-Instruct, at the three context lengths. Bold marks the better of each base/+RESCUE pair, per column.

Method	niah_single_2	niah_multikey_1	niah_multikey_2	niah_multikey_3	qa_hotpotqa	**Avg.**
Context length 16K						
Dense	100.0	99.0	100.0	99.0	61.0	91.8
LaProx	96.0	99.0	99.0	0.0	57.0	70.2
+ RESCUE	**97.0**	99.0	**100.0**	0.0	57.0	70.6
H2O (evict-once)	2.0	4.0	0.0	0.0	43.0	9.8
+ RESCUE	**29.0**	**53.0**	**55.0**	0.0	**54.0**	38.2
SnapKV	98.0	99.0	**97.0**	0.0	56.0	70.0
+ RESCUE	**99.0**	99.0	93.0	0.0	**57.0**	69.6
LAVa	99.0	97.0	**98.0**	0.0	56.0	70.0
+ RESCUE	**100.0**	**98.0**	97.0	0.0	56.0	70.2
R-KV	99.0	99.0	93.0	0.0	**58.0**	69.8
+ RESCUE	99.0	99.0	**94.0**	0.0	57.0	69.8
Context length 32K						
Dense	100.0	100.0	97.0	100.0	51.0	89.6
LaProx	100.0	100.0	**96.0**	0.0	50.0	69.2
+ RESCUE	100.0	100.0	95.0	0.0	50.0	69.0
H2O (evict-once)	2.0	5.0	0.0	0.0	39.0	9.2
+ RESCUE	**18.0**	**17.0**	**30.0**	0.0	**45.0**	22.0
SnapKV	99.0	100.0	79.0	0.0	47.0	65.0
+ RESCUE	99.0	100.0	**81.0**	0.0	**50.0**	66.0
LAVa	99.0	**100.0**	90.0	0.0	47.0	67.2
+ RESCUE	99.0	99.0	**93.0**	0.0	**49.0**	68.0
R-KV	100.0	**100.0**	90.0	0.0	49.0	67.8
+ RESCUE	100.0	99.0	**92.0**	0.0	49.0	68.0
Context length 64K						
Dense	100.0	99.0	96.0	99.0	50.0	88.8
LaProx	100.0	**99.0**	88.0	0.0	49.0	67.2
+ RESCUE	100.0	96.0	88.0	0.0	49.0	66.6
H2O (evict-once)	2.0	3.0	0.0	0.0	38.0	8.6
+ RESCUE	**19.0**	**13.0**	**12.0**	0.0	**42.0**	17.2
SnapKV	99.0	98.0	49.0	0.0	43.0	57.8
+ RESCUE	99.0	98.0	**51.0**	0.0	**46.0**	58.8
LAVa	99.0	98.0	**73.0**	0.0	46.0	63.2
+ RESCUE	99.0	98.0	68.0	0.0	46.0	62.2
R-KV	97.0	99.0	48.0	0.0	48.0	58.4
+ RESCUE	**100.0**	99.0	**58.0**	0.0	48.0	61.0

B EXPERIMENTAL DETAILS

We evaluate RESCUE on Meta-Llama-3.1-8B-Instruct, Mistral-7B-Instruct-v0.3 and Qwen3-8B across all 16 LongBench tasks using the official task metrics. Every analysis beyond Table 1 — the mechanism study, the selector quadrants, the headroom and the overhead measurements — is on Llama alone; Mistral and Qwen each carry the full 16×5 grid and the dense row, and are what the transfer claim rests on. Policy-specific residual scorers are trained on Natural Questions and arXiv documents disjoint from LongBench, with oracle future-attention importance computed from up to 256 future queries. The three backbones retain different numbers of those documents — 499 on Llama, 443 on Mistral and 780 on Qwen — because caching drops a document whose teacher-forced continuation is too short to form a horizon, and where that falls depends on the tokenizer.

All methods retain 128 prompt KV positions *per layer*; across the L layers this is the model-wide budget written $B_{\text{total}} = 128L$ in Table 1. Each base policy keeps its own protected recency window and contests the remaining slots, and RESCUE inherits that window unchanged so that a declined correction reproduces the base policy exactly: 32 protected positions for SnapKV and LaProx, 64 for H2O, R-KV and LAVa (Appendix B.1). We augment SnapKV, LaProx, LAVa, R-KV, and H2O with RESCUE, and compare against the future-aware ForesightKV and LookaheadKV baselines. Model checkpoints, prompt templates, and decoding configurations are kept identical across methods. Additional implementation details are provided in Appendix B.1.

B.1 PROTOCOL AND HYPERPARAMETERS

Why the objective needs the selector. A (document, layer) pair with $\mathcal{R} = \emptyset$ has no defined target and is skipped in training, so the scorer's output is a distribution over candidates, renormalized whether or not the base policy missed anything. It has no way to express "nothing needs recovering", which is exactly the failure the ablation measures on the strongest base policy.

Teaching the objective that state does not remove the need for the probe. Training the $\mathcal{R} = \emptyset$ pairs against a uniform target instead of skipping them, everything else fixed, gives a scorer that can represent "nothing to recover"; applied unconditionally on SnapKV it reaches $+0.91$ over the 16 tasks, with a worst cell of -5.94 and three cells losing at least a point. Gating the same scorer with the probe restores it to $+1.83$, a worst cell of -0.86 and no cell losing a point — the same shape the skipped-pair scorer shows, $+1.11$ ungated against $+1.71$ gated. The missing state can be learned, and the probe still earns its cost either way; what changes is which scorer the selector is gating, not whether it is needed. On the mean the no-rescue scorer is $+0.12$ ahead of the one we ship and its tail is 0.64 deeper, on one base policy; we did not repeat it on the other four.

Scope of a selection. The rescue-set target is defined at $B = 128 - 4 - 32 = 92$ selectable slots (Section 4.1), while each base policy's own protected window governs inference. The policies miss sufficiently different entries to justify one scorer each: pairwise rescue-set Jaccard runs from 56.9% to 94.3%.

We evaluate on Meta-Llama-3.1-8B-Instruct, Mistral-7B-Instruct-v0.3 and Qwen3-8B with all 16 LongBench tasks and the official task metrics, under OpenCompass; the RULER (Hsieh et al., 2024) runs of Appendix J use the same harness and the same per-layer budget. Generation, prompt templates and decoding configuration are identical across every method compared, and across the three backbones; each backbone has its own five residual scorers, trained by the same recipe on its own cached features, with no per-model tuning.

Budget and protected windows. All methods retain 128 prompt KV positions per layer. Compression happens once, after prefill; no entry is evicted during decoding. The maximum sequence length is each backbone's own trained position range — $131{,}072$ for Llama-3.1-8B-Instruct, $32{,}768$ for Mistral-7B-Instruct-v0.3 and $40{,}960$ for Qwen3-8B — and documents longer than that are truncated head and tail rather than extrapolated. The limit is a per-model setting rather than one shared constant because running Mistral past its own range degraded its eviction scores on the one task where that happened. Qwen3-8B is run with `enable_thinking=False` throughout, so all three backbones answer in a single non-reasoning pass. Each base policy protects its own recency window and lets the remaining slots be contested by the score, and RESCUE reuses that window unchanged so that a declined correction reproduces the base exactly:

Base policy	Protected window	Contested slots
SnapKV	32 (observation window)	96
LaProx	32 (its published convention)	96
H2O	64 (recency)	64
R-KV	64 (recency)	64
LAVa	64 (recency)	64

The supervision target is defined at a separate, policy-independent size $B = 128 - 4 - 32 = 92$, the budget net of the sink positions and the observation window; this is a property of the training target, not of the inference-time allocation.

Hardware and software. Every number in this paper was produced on one NVIDIA RTX PRO 6000 Blackwell Server Edition (96 GiB), with PyTorch 2.13, CUDA 13.0 and Transformers 4.53 in `eager` attention, which the eviction hooks require in order to read attention weights back from the forward pass. Latency and peak-memory measurements hold the card exclusively; accuracy runs do not, since they do not depend on it.

Base policy hyperparameters. SnapKV: observation window 32, pooling kernel 7, average pooling. R-KV: window size 8, retain ratio 0.1, mixing coefficient $\lambda_{\mathrm{R-KV}} = 0.07$ (distinct from the correction's λ), kernel size 7, retain direction *last*.

The five differ in how a layer's budget is divided, and RESCUE inherits that division rather than imposing one: the correction changes the *score* a policy ranks on and nothing else, which is what makes $\lambda = 0$ reproduce the base policy exactly. SnapKV and H2O spend a uniform per-head budget; LaProx uses `global_layer` allocation, ragged across layers, and its RESCUE arm runs the same; LAVa derives a per-layer budget from score entropy and spends it from one pool shared by all heads, so head counts differ within a layer; R-KV takes a per-kv-head top-k with the same count in every head. For LAVa and R-KV the correction is folded into that policy's own state class, so its allocation is the base policy's by construction. Combination is `scale_additive` at $\lambda = 1$ when the selector accepts.

Selector. The fidelity selector generates a single probe token from the uncompressed cache immediately after prefill and compares $\lambda \in \{0, 1\}$, adopting the corrected cache only if it keeps the output distribution closer to the dense model's; ties resolve to $\lambda = 0$.

Chat template. Llama-3.1's chat template is applied to all tasks except TREC (Li & Roth, 2002), TriviaQA (Joshi et al., 2017), SAMSum (Gliwa et al., 2019), LCC (Guo et al., 2023) and RepoBench-P (Liu et al., 2024), which LongBench specifies as completion-style; applying it to those tasks depresses their scores, and the exclusion is enforced in the harness rather than left to the invocation. RULER carries no such exception: its own configuration assigns the backbone's chat template to every task, and we follow it.

Deviations from published settings. Two baselines cannot be run exactly as published at this operating point, and we state both.

ForesightKV is run with its recency window reduced from its published 128 to 64. The published value is incompatible with a 128-token budget: the window consumes the entire per-layer budget and the scorer receives no slots at all, producing degenerate output (Qasper 3.55). We therefore report the budget-compatible configuration in the main table and the published configuration's score here, so that both are on record. Neither is a tuned setting — 64 is the largest window that leaves the scorer any slots at a budget of 128.

H2O is a decode-time policy in its original form: it re-ranks heavy hitters as generation proceeds and evicts continuously. Every method in this paper is evaluated under one protocol — compress once at the end of prefill, never during decoding — because the fidelity selector requires the dense prompt cache to exist at that moment, and a comparison in which only some policies kept evicting during generation would not isolate the correction. The H2O variant here therefore accumulates the same attention statistic over the prompt and applies it at that single eviction point. This is a faithful port of H2O's *scoring rule* but not of its *eviction schedule*, and its scores should be read as those of the evict-once variant rather than of the published method.

B.2 Training Details

Policy-specific residual scorers are trained on 499 documents disjoint from LongBench: 243 from Natural Questions (Kwiatkowski et al., 2019) and 256 from arXiv (Cohan et al., 2018). The NQ portion is what survives feeding 520 raw NQ documents through the pipeline — rather more than half are discarded for having a continuation too short to define the target — so 520 is the number of raw documents read and 499 the number that produce training data. Oracle future-attention importance s^{GT} is computed from up to 256 future queries of the reference continuation. The arXiv portion shares a source domain with Qasper, the task RESCUE gains most on, so we checked for document reuse directly: over 8-gram shingles the largest Jaccard similarity between any training document and any of Qasper's 200 is 0.0008, a single shared shingle, and no pair exceeds 0.02. The two sets are disjoint at the document level. The seed-0 split holds out the first 40 documents for validation, leaving 459 for training.

Scorer. The scorer is an MLP with layer sizes $18 \to 32 \to 16 \to 1$ and ReLU activations, shared across all 32 layers. Its 18 inputs are 9 raw per-entry features together with their within-layer percentile ranks; the rank features are what let one scorer serve layers with very different score scales. Normalization statistics are computed over the first 12 training documents on layers $\{0, 8, 16, 24, 31\}$ only, with the standard deviation clamped at 10^{-3}.

Objective. Training is listwise: the loss is $-\sum_i t_i \log \mathrm{softmax}(\mathrm{logits})_i$ with t uniform over the rescue set $\mathcal{K}_{\mathrm{GT}} \setminus \mathcal{K}_{\mathrm{base}}$ at the target size B. This supervises the ranking of the entries the base policy missed rather than a pointwise regression onto future attention mass.

Optimization. Adam at learning rate 10^{-3}, one step per (document, layer) pair, 15 epochs. The document split is a fixed shuffle with seed 0, the first 40 documents held out to monitor convergence. The checkpoint is not selected on that split: every scorer is taken at epoch 15, fixed in advance, so no per-policy tuning enters the comparison. We use a single checkpoint per (base policy, backbone, budget) — one for each of SnapKV, LaProx, LAVa, R-KV and H2O, five per combination. That is 15 for Llama-3.1 across the three budgets and five each for Mistral and Qwen3; the policy-agnostic full-future control of Section 5.4 is a twenty-sixth, trained identically on $\mathcal{K}_{\mathrm{GT}}$.

Cost. The expensive part of the pipeline is the forward passes that capture query and key/value states; these are shared across base policies because the 18 features do not depend on the base policy. Features are cached once and replayed, so training a scorer for an additional base policy costs only the optimization above.

C The Observation Bottleneck, in Detail

Can this limitation be removed by a better scoring rule alone? We compare the rankings of SnapKV, LaProx, LAVa and R-KV, which score differently but read the same local observation window, by the Spearman correlation $\rho_{m,n} = \mathrm{Spearman}(s^m, s^n)$. Pairwise correlations run from 0.763 to 0.971 (LaProx–LAVa 0.971, SnapKV–LaProx 0.943, SnapKV–LAVa 0.940): despite differing in attention aggregation and value-aware contribution, policies reading the same evidence rank tokens very similarly.

Rank correlation is weak evidence on its own, since eviction depends only on the top-B boundary. We therefore measure the failures directly, as the fraction of oracle-critical entries that *both* policies fail to retain, normalized against what two independently failing policies would produce (Appendix G). The raw co-miss rate is 52.0–80.4%, but each policy individually misses 65–84% of the oracle set here, so most of that is arithmetic: the excess over independence is a lift of 1.03 to 1.37×. The failures are correlated, but modestly.

Where the lift is informative is in *which* policies share failures: it tracks what a policy can see rather than how it scores what it sees, and the per-pair figures are in Appendix G. That is the scope restriction Section 3.2 names, and the residual missed by only one policy (1.4–26.4%) is what makes the rescue set policy-specific and motivates training a separate scorer per base policy (Section 4.1).

Table 7: Accuracy against cost, Llama-3.1-8B-Instruct with SnapKV as the base. Scores are the 16-task LongBench mean; added logic is the median eviction- and selection-related latency per document on Qasper, measured as in Appendix L. Retained entries per layer is what the cache costs for the whole of decoding.

Method	Score	Added logic	Retained / layer
SnapKV, B=128	43.03	5.6 ms	128
SnapKV, B=256	45.70	5.4 ms	256
SnapKV, B=1024	**48.84**	5.3 ms	1024
RESCUE, B=128	44.74	277.2 ms	128
RESCUE, B=256	46.68	275.1 ms	256
RESCUE, B=1024	48.83	274.2 ms	1024

Together with the complementarity above, this suggests a residual view of future-aware eviction: rather than replacing the base policy with a predictor of the entire future-importance distribution, it suffices to focus on the future-critical entries the base policy misses. This motivates RESCUE.

D Accuracy Against Cost

Three components could be tuned for accuracy; we report each against its cost instead. A fourth, the direction of the divergence, is fixed rather than swept: p_{dense} is the reference the candidates are judged against, and $D_{\mathrm{KL}}(p_{\mathrm{dense}} \parallel p_{\mathrm{pruned}})$ penalizes a candidate that puts little mass where the dense model puts much — the failure eviction causes — while leaving mass the dense model does not use unpenalized. The shipped configuration already spends more added logic than any baseline we compare with — 276.1 ms against ForesightKV's 134.7 ms — so these are choices about how much further to spend, not about staying inside a budget. Longer probes genuinely help — lengths 4 and 8 beat one token by $+0.40$ and $+0.43$ over 32 cells ($p = 0.002$, $p = 0.001$) — but length 2 already costs 330.4 ms against length 1's 276.1 ms, and each probe token delays compression by a decode step, charged against the memory argument the method rests on. A finer grid $\lambda \in \{0, 0.5, 1, 2\}$ buys $+0.07$ with an inconsistent sign at roughly 507 ms. Capping the correction at k contested slots does not pay either: the unrestricted correction is best throughout (SnapKV/Qasper $+2.24$ at $k = 16$ against $+7.31$), because bounding the damage bounds the benefit, whereas the selector rejects the correction outright where it would hurt. We ship probe length 1, two candidates and no cap, forgoing $+0.43$ points (Appendices O.1, O.3, O.4).

D.1 Against a Larger Cache

The comparison a deployment actually faces is not RESCUE against another eviction rule at the same budget, but RESCUE against spending the same resources on a larger cache. Table 7 puts both on the same axes.

RESCUE's overhead is paid once; a larger cache is paid per decoded token and per sequence held concurrently, so the two orderings need not agree. Table 8 measures the decode side directly, building a cache of the requested width and decoding against it, since the evaluation harness generates one sequence at a time and cannot be used for this. At batch one the two budgets are indistinguishable (15.18 against 15.21 ms per token), and RESCUE's 277 ms would need 1,600 generated tokens to repay. From batch four on the gap opens and the break-even falls to roughly 2,000 tokens: 2,199 at batch four, 1,908 at eight, 1,976 at 64 and 2,090 at 128. The accuracy ordering does not reverse with it — SnapKV at $B = 256$ is still $+0.96$ ahead — so what the measurement establishes is the regime where the smaller cache pays for itself in time and memory, not one where it also wins on score.

Generation length dilutes the same cost from the other side. Run end to end at batch one on three LongBench tasks whose outputs differ by an order of magnitude, RESCUE's added logic is 27.4% of total latency on Qasper (median 25 generated tokens), 3.3% on GovReport (425) and 3.0% on MultiNews (512), against 0.1–0.7% for SnapKV at either budget. The share falls by nearly an order of magnitude while the absolute cost stays near 300 ms, which is what "paid once after prefill" means

Table 8: Decode throughput on Llama-3.1-8B-Instruct at a 5K-token cache width, by batch and per-layer budget. $B = 5120$ is the uncompressed cache. Break-even is the generated length at which RESCUE's 277 ms one-time cost is repaid against $B = 256$.

Batch	B=128	B=256	B=1024	B=5120	**Break-even**
		tokens/s			tokens
1	65.9	65.8	64.8	54.3	11,620
4	256.2	248.2	215.6	112.7	2,199
8	483.9	452.2	366.7	142.2	1,908
32	1514.4	1300.3	627.5	175.4	2,547
64	2418.3	1806.1	741.6	183.6	1,976
128	3196.4	2245.3	807.6	OOM	2,090

in practice; it does not by itself make RESCUE faster end to end at batch one, where SnapKV at $B = 256$ still finishes MultiNews in 9,920 ms against 10,483.

At batch 128 the uncompressed cache no longer fits and $B = 1024$ runs at a quarter of $B = 128$'s throughput, so the aggressive-compression regime the paper targets is also the regime where eviction matters most for serving. Whether the correction is worth its one-time cost there is the question Table 7 answers on accuracy and this one on time.

Read across the two blocks, raising the budget is the better trade wherever the budget is free to raise: SnapKV at $B = 256$ scores $+0.96$ above RESCUE at $B = 128$ and pays 272 ms less per document.

What the table also shows is where that stops being true. RESCUE's overhead is flat in the budget — 277.2, 275.1, 274.2 ms — because the cost is building and replaying the candidate caches, which scales with the prompt rather than with how much of it is kept. The gain is not flat: $+1.71$ at $B = 128$, $+0.98$ at $B = 256$, -0.02 at $B = 1024$. The same fixed price buys less and less, which is what fixes the operating point Section 5.4 states.

Section 5.4 notes that the two costs are not paid on the same schedule. They also come out of different budgets: 272 ms of time against twice the cache memory, which a deployment may not be free to trade against each other at all.

E Scope of the Evidence, in Detail

Each claim above rests on a specific control; here is the one each still lacks.

Statistical power on the target comparison. The residual target is ahead in both regimes and on four of five base policies, but 80 paired cells cannot resolve a margin of $+0.22$ (41W/28L, $p = 0.148$). Retraining the scorer under different seeds says why, and the answer is not only sample size. Re-running the recipe for H2O and R-KV with two further seeds — a different document split and initialization, everything else fixed — moves the task-averaged gain by a standard deviation of 0.49 and 0.34 respectively (Appendix P). A margin of $+0.22$ is smaller than the spread a single design choice already produces, and because one checkpoint serves all 16 tasks that spread does not average away over cells. More cells alone will therefore not settle this comparison; separating a difference this size needs the seed itself treated as a repeated factor.

How well one token predicts the outcome. Over all 18,750 evaluation documents the selector picks the retrospectively better cache on 55.6% of those where the arms differ, and the KL margin correlates with the realized score difference at $\rho = +0.11$. It still recovers 28.4% of the oracle headroom — its $+2.04$ per decisive document against the $+7.21$ a perfect choice would return on the same documents (Appendix N.3) — because accuracy is the wrong statistic: rejecting costs nothing, and the accepts it gets right are larger than the ones it gets wrong (Figure 4, right). Nor is low accuracy the same as choosing at random — a gate accepting the same share uniformly at random returns $+0.17$ points per document against the selector's $+2.04$ (Appendix N.3). The pooled figure also averages two regimes: on the fifth of documents where its own margin is largest the selector is right 63.6% of the time and worth $+6.00$ points per document, against 52.5–55.5% and at most $+2.10$ on the rest (Appendix N.2). It is near chance where the probe carries no information and

better than chance where it does, and the second regime is where the large score differences are. Its errors are concentrated: on TREC the decision is essentially free and the selector reaches the oracle exactly, while on the code tasks one position carries almost no information about an edit-similarity score (-45.9% of headroom on LaProx/LCC). The gap to the oracle selector (45.05 against 47.32) is the clearest headroom here.

Context length. Both axes are measured. The budget axis runs at $B \in \{128, 256, 1024\}$ (Section 5.4), where the gain falls to zero on the stronger base policies at the widest setting. The length axis runs to 64K on RULER (Appendix J), and there the pooled gain *falls* with length, $+5.72$ to $+2.12$. What the two axes share is not length but headroom: on both, the correction returns what the base policy left behind, and the four policies that stay functional on RULER gain more as they themselves lose more ($+0.05$, $+0.45$, $+0.50$). Length is the weaker control on headroom of the two, so the budget is the axis the operating point should be read on. Whether a benchmark exists where length alone drives the base policies down as far as a tight budget does is open.

Oracle definition and generality. s^{GT} is future attention mass, not measured causal contribution; validating it against deletion-based influence would establish whether the target the scorer is trained on is the target that matters. The results are on three backbones of similar size, so scale is untested, and whether a scorer transfers between base policies is open — poor transfer supports the policy-conditioned framing, good transfer would reduce the number of scorers a deployment needs.

F Base Eviction Policies and Future-Aware Methods

F.1 Observed Importance

Observed importance estimates KV importance using model behavior that has already been observed by the time eviction is performed. For example, SnapKV selects important prefix KV entries using attention measured over an observation window near the end of the prompt, while H2O maintains heavy-hitter KV entries based on accumulated attention. In its original formulation H2O evicts *during* decoding, re-ranking heavy hitters as generation proceeds; the variant evaluated here accumulates the same attention statistic over the prompt and evicts once after prefill, so that it is compared under the same protocol as every other policy (Appendix B.1). LaProx further incorporates the effect of the value representation and output projection, in addition to observed attention, to estimate token contribution.

A second line keeps a scoring rule fixed and reallocates the budget across heads or layers — AdaKV (Feng et al., 2025), PyramidKV (Cai et al., 2025b) and CAKE (Qin et al., 2025). We do not augment these, because they are orthogonal to the question here: they change how many slots a layer or head receives, not which entries fill them, so a residual correction applies to whichever scoring rule they allocate for rather than competing with the allocation itself.

Although these methods differ in their scoring rules and eviction timing, they share a common principle: future importance is inferred from evidence that has already been observed. This enables efficient eviction without additional future generation, but it also limits these methods to dependencies that are visible before or at the eviction point. Dependencies that emerge only during subsequent generation cannot be directly observed.

F.2 Predicted Future Importance

Predicted future importance instead estimates how important each KV entry will become during a response that has not yet been generated. These approaches either predict future utility from KV states and attention-related features available at inference time, or construct a surrogate representation intended to approximate the behavior of the true future response.

For example, KVP learns a KV-ranking policy using future decoding utility as supervision, while ForesightKV trains a lightweight scorer to predict long-term importance derived from future attention. LookaheadKV approximates the importance distribution induced by the true response using learnable lookahead tokens together with LoRA. Other approaches first generate a surrogate response and use its attention patterns as a proxy for future importance.

Despite their different implementations, these methods share the goal of introducing an additional estimate of future dependencies that cannot be directly recovered from observed evidence alone. Appendix G examines whether observed importance and predicted future importance identify the same KV entries, or instead capture complementary information.

G Set Definitions and Co-Miss Normalization

This appendix gives the set definitions and the normalization behind the complementarity and observation-bottleneck results of Section 3.

Position is not the distinguishing statistic. The three signals do emphasize different regions: over 200 Qasper documents the mean normalized position is 0.102 for observed importance against 0.634 for the oracle and 0.673 for the predicted signal. Position is a coarse summary, though, and what the argument needs is whether the signals recover different *entries*; that is what the decomposition below measures.

G.1 Exclusive Recall

We decompose the oracle set into the entries each signal uniquely recovers:

$$\mathcal{K}_{\text{obs-only}} = \mathcal{K}_{\text{GT}} \cap (\mathcal{K}_{\text{obs}} \setminus \mathcal{K}_{\text{pred}}),$$
$$\mathcal{K}_{\text{future-only}} = \mathcal{K}_{\text{GT}} \cap (\mathcal{K}_{\text{pred}} \setminus \mathcal{K}_{\text{obs}}).$$

For methods m and n, we measure ranking agreement using Spearman correlation:

$$\rho_{m,n} = \text{Spearman}\,(s^m, s^n).$$

G.2 Co-Miss and Its Normalization

A high global rank correlation is, on its own, weak evidence for this claim: eviction depends only on the top-B boundary, and small rank differences near the cutoff can still produce materially different retained sets. We therefore measure the failures directly. For two policies m and n we compute the fraction of oracle-critical entries that *both* fail to retain,

$$\text{CoMiss}_{m,n} = \frac{|(\mathcal{K}_{\text{GT}} \setminus \mathcal{K}_m) \cap (\mathcal{K}_{\text{GT}} \setminus \mathcal{K}_n)|}{|\mathcal{K}_{\text{GT}}|},$$

which asks whether the two policies are blind to the *same* entries rather than merely ranking them similarly. Across the five base policies, 52.0–80.4% of oracle-critical entries are missed by both members of a pair.

That number needs normalizing before it means anything. Each policy individually misses 65–84% of the oracle set at this budget, so two policies failing independently would already co-miss 44–70% by arithmetic alone. Against that baseline the excess is

$$\text{Excess}_{m,n} = \text{CoMiss}_{m,n} - \frac{|\mathcal{K}_{\text{GT}} \setminus \mathcal{K}_m|}{|\mathcal{K}_{\text{GT}}|} \cdot \frac{|\mathcal{K}_{\text{GT}} \setminus \mathcal{K}_n|}{|\mathcal{K}_{\text{GT}}|},$$

which is $+1.3$ to $+16.2$ points, a lift of 1.03 to 1.37 over independence. The failures are correlated, but modestly: most of the raw co-miss rate is a consequence of how much every policy misses, not of shared evidence.

The lift is nonetheless informative about *which* policies share failures. It is highest among the observation-window methods that read the same local evidence (SnapKV–R-KV 1.32, R-KV–LAVa 1.37) and lowest for every pair containing H2O (1.03–1.06), the one policy here that accumulates attention over the whole prompt rather than a recent window. Evidence scope, not scoring formulation, is what separates them.

H Mechanism Measurements

This appendix expands the mechanism analysis of Section 5.3.

Table 9: Mechanism measurements per base policy ($\lambda = 1$, correction always applied), against that policy's own downstream gain under the full method.

Base	ΔCoverage	Rescue recall	Downstream Δ
H2O (evict-once)	+25.7 pt	46.3%	+2.51
LAVa	+14.1 pt	30.6%	+1.09
SnapKV	+11.5 pt	28.7%	+1.71
LaProx	−4.1 pt	23.4%	+0.81
R-KV	−20.8 pt	0.7%	+0.27

H.1 Positional Mismatch Is a Weak Predictor

Section 3 showed that observed and oracle selections sit in different regions of the context, which suggests using the positional gap $D_{\text{mean-pos}} = |\mu_{\text{base}} - \mu_{\text{GT}}|$ as a coarse indicator of which tasks should benefit. Measured against the realized gains it is not one: $\rho = 0.113$ over the same 74 cells, against 0.476 for $\Delta\text{Coverage}$. Tasks with nearly identical positional gaps land at opposite ends of the gain distribution — TREC ($D = 0.220$) gains $+4.10$ while LCC ($D = 0.208$) loses 0.55. We report this because the positional analysis is what motivated the method, and it would be easy to present it as though it also explained the outcome. It does not; what a selection *contains* predicts the outcome, where it *sits* does not.

H.2 Per-Policy Mechanism Measurements

The table is the mechanism behind the ablation of Table 2: unconditional correction on R-KV costs 0.70 points and its worst cell loses 8.50, because a policy that was already retaining the oracle set has nothing to gain and entries to lose. The two measurements agree — the correction is useful in proportion to how much the base policy was missing, and the selector is what keeps it from being harmful where there is little to recover.

I Budget-Matched Complementarity

At the union's own size the predicted signal alone reaches 60.2% oracle coverage against the union's 54.7%, so mixing observed evidence in is not free. That comparison does not settle the design question, for two reasons. Oracle coverage counts entries, and which entries a model can actually use is not settled by counting them: the same comparison run end to end reverses, with an augmented observed policy ahead of both future-aware methods on all five base policies (Appendix A.1). And the premise of this paper is a system that has already committed to an observed policy, for which the question is not which signal to buy but what the one it already runs is throwing away.

The exclusive-recall result of Section 3 is budget-matched by construction, but the union coverage reported alongside it is not: the union of two top-B sets may hold up to $2B$ entries. To establish that combining the two signals helps under a fixed cache, we select a single top-B set from a normalized mixture

$$s_i(\alpha) = (1-\alpha)\,\tilde{s}_i^{\text{obs}} + \alpha\,\tilde{s}_i^{\text{pred}}, \qquad \alpha \in [0,1],$$

where $\tilde{s}$ denotes percentile-rank normalization, and report oracle coverage of the resulting B-entry selection against the $\alpha = 0$ and $\alpha = 1$ endpoints. The observed signal is SnapKV's own score; the predicted signal is the policy-agnostic full-future scorer of Section 5.4, trained on $\mathcal{K}_{\text{GT}}$. Measured over 293 documents from 15 of the 16 tasks, all but PassageRetrieval-en (Table 10).

How that split moves with the budget is worth recording separately, because it is the one place where our measurements speak to an operating point other than $B = 128$. Figure 1 (Section 3) decomposes the oracle-critical entries on Qasper by which signal recovers them, at four budgets. Two things change together as the budget tightens. The share both signals recover falls from 45.0% at $B = 1024$ to 21.7% at $B = 128$, and the share neither recovers rises from 23.9% to 38.6%. In other words the aggressive-compression regime is precisely the one in which a base policy leaves most on the table, which is the regime a correction is worth paying for. Here the observed signal is the mean percentile rank of the five base policies and the predicted signal that of ForesightKV and LookaheadKV, so the split is a property of the two *families* rather than of any one method. The

Table 10: Oracle coverage under a fixed budget of B selectable slots per layer. The union row is the quantity a naive complementarity argument would report; the two rows beneath it give observed-only and predicted-only selections of that same size, which is what the comparison requires.

Selection	$\lvert\mathcal{K}\rvert$	**Oracle coverage**
Observed only (α=0)	B	30.1%
Predicted only (α=1)	B	47.6%
Best mixture (α^*=0.875)	B	**49.2%**
Union (not budget-matched)	$1.64B$	54.7%
Observed only, matched to $\lvert$union$\rvert$	$1.64B$	39.3%
Predicted only, matched to $\lvert$union$\rvert$	$1.64B$	**60.2%**

downstream gain follows the same direction: Section 5.4 measures it at $B = 256$ and $B = 1024$ as well, where it falls from $+1.28$ to $+0.88$ to $+0.28$.

J RULER AT LONG CONTEXT

LongBench prompts average 5K tokens, so the budget axis of Section 5.4 and the length axis are confounded there: at $B = 128$ the compression ratio is about $40\times$, and nothing in that benchmark says what happens at $500\times$. RULER separates them. We run its five retrieval tasks — single-needle retrieval (niah_single_2), the three multi-key variants (niah_multikey_1–3) and multi-hop QA (qa_hotpotqa) — on Llama-3.1-8B-Instruct at 16K, 32K and 64K, holding the per-layer budget, the protected windows, the scorer checkpoints and the selector fixed at the settings of Appendix B.1; nothing is retrained or retuned for length. These are the tasks on which the rescue set is well defined: a specific span carries the answer, so an entry either belongs in the retained set or does not.

The prediction under test is the paper's own. Section 5.4 shows the correction is worth more the more the base policy must drop, measured by shrinking the budget; lengthening the input tightens the same ratio from the other side. If the residual view is right about the mechanism rather than about one benchmark, the gain should grow with length. It does not, and the reason is informative: length and budget do not move the base policies by the same amount.

At 16K, 32K and 64K the pooled gain is $+5.72$, $+2.92$ and $+2.12$ points, falling with length rather than rising. The pooled figure is set by H2O, which keeps under a tenth of the dense score at every length (9.80, 9.20 and 8.60 against 91.80, 89.60 and 88.80); RESCUE returns $+28.40$, $+12.80$ and $+8.60$ of that, and the recovery shrinks because what is still reachable from a collapsed cache shrinks with length.

Across the four policies that stay functional the gain does rise with length — $+0.05$, $+0.45$ and $+0.50$ — and it rises as those policies lose ground: SnapKV falls from 70.00 to 57.80 over the three lengths and its gain goes from -0.40 to $+1.00$; R-KV falls from 69.80 to 58.40 and goes from 0.00 to $+2.60$. LaProx, which loses least (70.20 to 67.20), is helped only at 16K ($+0.40$) and slips at the two longer settings. The mechanism holds on the axis the residual view actually names — the correction is worth what the base policy left behind — while length turns out to be a weak control on that quantity: only two of the five lose under 5 points over a fourfold increase, and one of those is H2O, which starts at 9.8 and has little left to lose.

The selector grows more conservative as the input lengthens, declining the correction on 60.1% of documents at 16K, 64.5% at 32K and 69.4% at 64K. Cells losing at least a point number 3 of 25 at 16K, 3 of 25 at 32K and 2 of 25 at 64K, and the worst single cell costs 5.0 points. The per-policy ordering of Section 5.2 survives the change of benchmark: H2O gains most and the strong observed policies gain least.

Table 11: RULER at $B_{\text{total}} = 128L$ on Llama-3.1-8B-Instruct, averaged over the five tasks at each context length. Each base policy is paired with the same policy under RESCUE. Per-task scores are in Appendix A.3.

Base policy	**16K**	**32K**	**64K**
Dense (no eviction)	91.80	89.60	88.80
LaProx	70.20	69.20	67.20
+ RESCUE	70.60	69.00	66.60
H2O (evict-once)	9.80	9.20	8.60
+ RESCUE	38.20	22.00	17.20
SnapKV	70.00	65.00	57.80
+ RESCUE	69.60	66.00	58.80
LAVa	70.00	67.20	63.20
+ RESCUE	70.20	68.00	62.20
R-KV	69.80	67.80	58.40
+ RESCUE	69.80	68.00	61.00
Mean Δ	$+5.72$	$+2.92$	$+2.12$
Mean Δ, excluding H2O	$\mathbf{+0.05}$	$\mathbf{+0.45}$	$\mathbf{+0.50}$

K Headroom Against an Oracle Future Signal

This appendix supports the headroom result of Section 5.3. The mechanism measurements in the body say what the correction changes, not how much was available to change. To bound the latter, we replace the learned scorer with the true future attention of the continuation — an oracle future-importance signal — at the same 128-token budget, holding the allocation, protected set and combination rule fixed. Because the oracle target never references a base policy, one run per task suffices. All 16 tasks are covered. We define a task's *headroom* as the oracle score minus the mean score of the five base policies, and its *recovery* as RESCUE's mean gain over that headroom.

Headroom is small on most of LongBench and very large on a few tasks (Table 12): mean 5.30 points, median 3.40, range 0.14 (PassageCount) to 19.80 (TREC) and 16.07 (Qasper). Over the tasks with more than a point of headroom, RESCUE recovers 24.4% of it on average (26.4% median). PassageCount is excluded from that average because its headroom of 0.14 points makes the ratio uninformative rather than favorable.

Headroom predicts the size of the gain only weakly. The rank correlation between a task's headroom and RESCUE's gain on it is $\rho = +0.244$ ($n = 16$) — positive, but far from the near-monotone relationship one might expect, and at this sample size not a result. The tertile contrast is the clearer statement: the bottom third of tasks by headroom (1.56 points available) gain $+0.50$, while the top third (10.98 available) gain $+2.55$. What breaks the correlation is a set of tasks where the headroom is real and the method does not reach it: GovReport has 7.10 points available and gains $+0.11$, MultiNews has 4.46 and gains $+0.07$, and the two code-completion tasks have 4.64 (LCC) and 5.12 (RepoBench) available while RESCUE *loses* 0.55 and 0.13. On GovReport and MultiNews this is not the selector declining: the correction applied unconditionally, which uses no probe, is worth -0.02 and -0.27 there. These are the same tasks on which the single-token fidelity probe is least informative (Section 5.5), so the correction is neither reliably proposed nor reliably declined there. Headroom is therefore necessary for a large gain but not sufficient for one: of what an oracle future signal would buy at this budget, the learned correction takes 24.4%.

Table 12: Headroom at a 128-token budget: an oracle future-importance signal against the mean of the five base policies, per task, with RESCUE's mean gain over the same five. Recovery is RESCUE's gain as a fraction of headroom; it is omitted for PassageCount, whose 0.14-point headroom makes the ratio meaningless. The rank correlation between headroom and gain is $\rho = +0.244$.

Task	Oracle	Base (mean)	Headroom	RESCUE Δ (recovery)
TREC	72.00	52.20	19.80	+4.10 (20.7%)
Qasper	42.84	26.77	16.07	+5.65 (35.2%)
GovReport	27.90	20.80	7.10	+0.11 (1.6%)
MultiFieldQA-en	55.81	48.97	6.84	+3.02 (44.2%)
RepoBench-P	57.76	52.64	5.12	−0.13 (−2.6%)
LCC	64.37	59.73	4.64	−0.55 (−11.9%)
MultiNews	24.67	20.21	4.46	+0.07 (1.6%)
2WikiMQA	49.82	46.37	3.45	+0.91 (26.4%)
NarrativeQA	29.43	26.08	3.35	+3.03 (90.6%)
MuSiQue	31.95	28.65	3.30	+1.47 (44.7%)
SAMSum	42.98	40.14	2.84	+0.29 (10.3%)
TriviaQA	91.96	89.38	2.58	+0.95 (36.6%)
HotpotQA	57.18	55.27	1.91	+0.74 (38.6%)
QMSum	24.73	23.03	1.70	+0.05 (2.9%)
PassageRetrieval-en	100.00	98.55	1.45	+0.40 (27.6%)
PassageCount	11.71	11.57	0.14	+0.37 (–)
Mean	–	–	5.30	24.4% recovered

L Latency Protocol

Latencies are measured on Llama-3.1-8B-Instruct at an average prompt length of 5K tokens, on Qasper over 200 documents, reporting the median per document on an exclusive GPU. Medians, not means: the distribution has a long right tail from the longest documents, and the mean of a RESCUE run sits about 20 ms above its median. "Added logic" counts eviction- and selection-related work only, excluding prefill and generation; for every row but LookaheadKV it is the wrapper-level interval around scoring and pruning, which is where each policy's own machinery runs.

The peak is not carried into generation: over the same 200 documents the KV cache RESCUE decodes with is its base policy's, 18.6 against SnapKV's 19.0 MB per sequence at $B = 128$ and 130.6 against 130.6 at $B = 1024$. But it is not free either, and it grows with the prompt. Table 13 repeats the measurement on RULER at three lengths with stage-level allocation logging. Two things come out of it.

First, the excess over the base policy is not the candidate caches. At 64K the allocation already stands at 58.8 GiB when the selector is entered, and building both candidates across all 32 layers adds 2.5 GiB more. To attribute the rest we ran one 64K document under both arms with every live CUDA tensor grouped by shape at the peak. The peak allocation is 63.1 GiB with RESCUE against 42.6 GiB for SnapKV alone, and the 20.6 GiB between them is one object: the per-layer prompt hidden states, 33 tensors of $(1, 65{,}020, 4096)$ in `bfloat16` at 0.50 GiB each, 16.4 GiB in total, which are live under RESCUE and absent under SnapKV. The scorer's nine raw features are read from them, and the released implementation holds all of them rather than consuming and freeing each layer's in turn. The remaining 4.2 GiB is the two candidate caches and the uncompressed prompt cache held until the selector finishes.

The largest single tensor at the peak is not part of the gap: a 15.5 GiB full-vocabulary logits tensor over every prompt position, which both arms allocate because the harness materializes logits at all positions rather than only the last. It is an artifact of the evaluation path, identical for the base policy, and a deployment that needs only the final position would not pay it. So the excess RESCUE is responsible for is an implementation choice in how the features are gathered, not a property of the method: it scales with the prompt rather than with the budget, and a per-layer feature pass would remove most of it.

Second, the added logic is linear in the prompt for SnapKV (376, 584, 942 ms) and quadratic for R-KV (1.6, 5.5, 22.1 s). The quadratic term is the base policy's own: R-KV builds an $n \times n$ redundancy

Table 13: RULER `niah_single_2` at $B = 128$, median over 100 documents. "Added" is eviction- and selection-related logic; "decode KV" is the cache generation actually runs against.

		16K	32K	64K
Added logic	SnapKV	8 ms	13 ms	23 ms
	+RESCUE	376 ms	584 ms	942 ms
	R-KV +RESCUE	1.6 s	5.5 s	22.1 s
Peak memory	SnapKV	22.1 GiB	29.4 GiB	43.9 GiB
	+RESCUE	30.7 GiB	42.0 GiB	68.3 GiB
Decode KV, either arm		16.5 MB	16.5 MB	16.4 MB

Table 14: Eviction and selection latency, ordered by cost. †LookaheadKV decides its eviction inside the patched attention rather than in the wrapper the other rows are timed in, so its row sums its own per-layer calls instead; that sum excludes the lookahead positions it adds to the prefill, and is therefore a lower bound on its added cost rather than a like-for-like figure.

Method	**Latency**
SnapKV	5.6 ms
LookaheadKV†	11.2 ms
ForesightKV	134.7 ms
RESCUE correction only	145.6 ms
RESCUE + 1-token selector	277.2 ms
dense probe	*20.9 ms*
two candidate caches	*253.8 ms*

block per layer. That block reads the query window and the keys, and the candidate λ changes neither — every candidate is built from a clone of the same dense cache — so the selector computes it once per document instead of once per candidate. The saving is exact rather than an approximation: an implementation that rebuilt the block per candidate measured 2.8, 10.5 and 43.5 s at these same three lengths and produced the same predictions on 900 documents of Qasper, TREC and LCC under R-KV, the policy whose ranking is the most sensitive to a change of this kind. What is left at 64K is 22 seconds of R-KV's own scoring rule per document, which the correction inherits rather than creates. The SnapKV rows are the control: they do not build that block, and they do not move.

Two cautions on reading these numbers. Per-token decode latency varies by 1–2 ms between runs of *identical* code and data on this machine, which is larger than the gap between any two arms we measured, so we do not read a per-token ordering off this measurement; the break-even lengths of Section 5.4 come from the decode-throughput benchmark of Appendix D.1 instead. What this measurement supports is that RESCUE's decode cost is not distinguishable from its base policy's.

M Correction and Selector, by Task

The selector's effect is concentrated in the tail. It cuts cells losing at least a point from 18 to 3, and its largest single rescue is H2O on TREC, where the correction applied unconditionally is worth -9.50 and the gated correction $+4.00$.

Per base policy. Over the 16 tasks the gain excludes zero on three of the five base policies — H2O $+2.51$ $[+1.17, +4.20]$, SnapKV $+1.71$ $[+0.79, +2.86]$ and LAVa $+1.09$ $[+0.22, +2.17]$ — against LaProx $+0.81$ $[-0.10, +1.91]$ and R-KV $+0.27$ $[-0.26, +1.00]$. The ordering is the coverage ordering: the policies whose intervals clear zero are the ones the mechanism measurements show leaving most unrecovered.

Section 5.4 reports the correction/selector split per base policy. The same split per task is below: *correction only* applies the correction to every document (λ=1, no probe), *selector* is the shipped method. Both are the mean over the five base policies, against each policy's own score.

Table 15: Per-task effect of the correction alone and of the selector, over the five base policies. A positive *selector − correction* means the selector adds to what the correction achieves; a negative one means it declines documents the correction would have won.

Task	Correction only	Selector	Selector − correction
Qasper	+6.85	+5.65	−1.20
TREC	−1.60	+4.10	+5.70
NarrativeQA	+3.36	+3.03	−0.33
MultiFieldQA-en	+3.41	+3.02	−0.39
MuSiQue	+1.57	+1.47	−0.09
TriviaQA	+0.77	+0.95	+0.18
2WikiMQA	+0.89	+0.91	+0.02
HotpotQA	+0.73	+0.74	+0.01
PassageRetrieval-en	+0.35	+0.40	+0.05
PassageCount	+0.37	+0.37	+0.00
SAMSum	−2.27	+0.29	+2.56
GovReport	−0.02	+0.11	+0.13
MultiNews	−0.27	+0.07	+0.34
QMSum	+0.20	+0.05	−0.15
RepoBench-P	−1.85	−0.13	+1.72
LCC	−2.25	−0.55	+1.70

Three regimes appear in Table 15. Where the correction is already right (Qasper, MultiFieldQA-en, NarrativeQA) the selector costs 0.33 to 1.20 points by declining documents it should have taken. Where the correction is wrong but the damage is visible to a one-token probe (TREC, SAMSum, and the two code tasks) the selector recovers +1.70 to +5.70. Where the correction has no signal at all (GovReport, QMSum, MultiNews) neither component has anything to work with, and the result sits within 0.15 of the base policy. The headline +1.28 is therefore not one mechanism applied sixteen times; it is a scorer carrying four tasks and a selector carrying four others.

N Selector Behaviour

The selector compares two divergences that can both be effectively zero. Accumulated in float32, a sum over a 150K-entry vocabulary carries an absolute error near 10^{-7}, so once the true divergence drops under that the comparison is between two rounding errors and the correction is accepted about half the time for no reason. How often that happens is a property of the backbone, not of the method:

Table 16: Share of documents on which *every* candidate's KL falls below 10^{-6}, with SnapKV as the base policy. Measured from the selector's own logs.

Task	Qwen3-8B	Llama-3.1-8B
PassageRetrieval-en	61.5%	0.0%
MultiFieldQA-en	42.7%	1.3%
HotpotQA	24.0%	1.6%
NarrativeQA	8.5%	0.0%
TREC	0.0%	0.0%

The rule that follows from Table 16 is to abstain: where every candidate falls below τ the selector returns $\lambda^* = 0$, so a document with no evidence is treated as a document with no correction rather than as a coin flip. The threshold is set above the float32 error, so the abstaining set contains every document on which the comparison is unreliable. Accumulating the divergence in float64 instead would remove the artifact at its source, at one extra log-softmax per candidate; the implementation supports it, but every number reported in this paper was produced in float32 with the abstention rule, which is the cheaper of the two and sufficient.

The threshold is not delicate. Raising τ enlarges the abstaining set considerably without moving the score, because the documents it adds are ones on which the correction was not going to matter either way:

Table 17: Qwen3-8B with SnapKV under three values of τ. The abstaining set grows by half again; the scores do not move.

	$\tau = 10^{-6}$	$\tau = 10^{-4}$	$\tau = 10^{-2}$
MultiFieldQA-en, abstained	$64/150$	$78/150$	$96/150$
MultiFieldQA-en, score	46.85	46.71	46.82
PassageRetrieval-en, score	97.00	97.00	97.00

What it changes, and what it does not. Table 17 sweeps τ over three values. On Qwen3-8B with SnapKV the rule moves the average over the 16 tasks from -0.16 to $+0.47$ against the base, almost all of it from four tasks: PassageRetrieval-en $91.00 \rightarrow 97.00$, MultiFieldQA-en $45.11 \rightarrow 46.87$, HotpotQA $53.74 \rightarrow 55.25$ and MuSiQue $26.62 \rightarrow 27.19$. The remaining twelve tasks move by at most 0.41.

Which results the rule was active for. The rule is the default in the released implementation. The Mistral-7B and Qwen3-8B results of Table 1 were produced with it; the Llama-3.1-8B results predate it, so we re-ran all 80 Llama cells with the rule enabled rather than argue the difference away. It is not material: 75 of the 80 cells are identical, and no base policy's 16-task mean moves by more than 0.03 — every move is downward, which is what the rule is for, since a document it abstains on is one the correction was winning by coin flip. The 80-cell mean falls by 0.01. The table reports the re-run.

The reason the two agree is that the rule is the *only* difference between the two code paths — the divergence is accumulated in float32 either way — so a cell on which no document abstains is bit-identical with the rule on or off. That invariant holds wherever we can check it: of the 80 Llama cells at $B = 1024$ for which both arms exist, all 36 with no abstention scored identically, as did all five such Mistral cells at $B = 128$.

On Llama at $B = 128$, where the paper's analyses live, 0.25% of probes fall below the floor and 61 of the 80 logged cells have none at all. The rest concentrate on 2WikiMQA and HotpotQA (16 probes each), MultiFieldQA-en (10) and MuSiQue (3), with one apiece on QMSum and Qasper. The five cells the full re-run moved are drawn from exactly that set — H2O/HotpotQA (-0.34, one degenerate probe), LAVa/2WikiMQA (-0.19, four), R-KV/2WikiMQA (-0.09, three), LAVa/MultiFieldQA-en (-0.05) and SnapKV/Qasper (-0.01).

We do not claim the rule is negligible in general. Where the probe is degenerate more often it is not: enabling it moves Mistral's PassageRetrieval-en by $+1.00$ on two base policies, and on Llama at $B = 1024$, where 2.4% of probes abstain, it moves 11 of 80 cells by at least 0.05 and one by 0.50, though the 80-cell mean moves only $+0.008$. The effect is concentrated exactly where the one-token probe carries no information, which is the regime the rule exists to handle.

N.1 Selecting Between Two Policies' Caches

This is the control behind the claim in Section 5.4 that the selector does not subsume the scorer. The selector machinery is unchanged — one probe token, the same $\mathrm{KL}(p_{\mathrm{dense}} \,\|\, p_{\mathrm{pruned}})$ criterion, the same λ-free decision — but the two candidate caches are those of two different policies rather than two settings of one.

We use ForesightKV rather than LookaheadKV as the future-aware candidate for a structural reason: LookaheadKV runs its own decoder path and never reaches the eviction machinery that builds candidate caches, so it cannot be a candidate under any selector of this form. ForesightKV is also the more expensive of the two comparators (134.7 ms against 11.2 ms, Appendix L), which makes this the harder control of the two. Substituting H2O for ForesightKV as the second candidate — the observed policy whose misses overlap SnapKV's least — gives $+0.27$, below ForesightKV's $+0.50$, so neither pairing approaches what the scorer reaches on the same base.

A cheaper use of the same probe. The selector already decodes one token against the uncompressed cache, and that token's attention over the prompt is a free sample of the future the base policy cannot see. Feeding it into SnapKV's own vote — no scorer, no second cache, no divergence

Table 18: Selecting between two policies' caches, against correcting one of them. All columns are LongBench scores at a 128-token budget on Llama-3.1-8B-Instruct. "CacheSel" applies the RESCUE selector to the SnapKV and ForesightKV caches; "RESCUE" corrects the SnapKV cache with the residual scorer under the same selector. The last column counts the documents on which CacheSel chose the ForesightKV cache.

Task	SnapKV	ForesightKV	CacheSel	RESCUE	Δ CacheSel	Fore. chosen
Qasper	25.15	21.88	24.26	**32.46**	−0.89	70 / 200
MultiFieldQA-en	49.44	33.36	48.58	**52.32**	−0.86	16 / 150
NarrativeQA	24.40	24.82	27.11	**28.10**	+2.71	71 / 200
HotpotQA	55.98	50.86	55.71	**57.21**	−0.27	35 / 200
2WikiMQA	46.22	44.74	46.74	**47.07**	+0.52	52 / 200
MuSiQue	28.08	25.53	28.04	**30.25**	−0.04	55 / 200
GovReport	18.68	22.32	**21.52**	20.36	+2.84	134 / 200
QMSum	22.93	22.19	22.64	**23.00**	−0.29	95 / 200
MultiNews	18.86	21.78	**21.36**	19.61	+2.50	163 / 200
TREC	47.00	41.00	48.50	**52.50**	+1.50	56 / 200
TriviaQA	89.33	87.10	89.23	**90.27**	−0.10	45 / 200
SAMSum	40.23	39.00	**40.62**	40.56	+0.39	53 / 200
PassageCount	12.00	12.00	12.00	11.80	+0.00	32 / 200
PassageRetrieval-en	98.00	95.50	**98.50**	**98.50**	+0.50	7 / 200
LCC	59.34	55.87	**59.44**	59.12	+0.10	204 / 500
RepoBench-P	52.83	48.92	52.24	**52.76**	−0.59	211 / 500
Average Δ vs. SnapKV	–	−2.60	+0.50	**+1.71**		
Win / loss vs. SnapKV	–	3 / 12	8 / 6	**13 / 3**		

— is the cheapest thing one can do with this probe, and on the 16-task mean it beats RESCUE: +1.95 against +1.71 over the same SnapKV base. The margin is one task. On TREC the probe row lands on the in-context labels a few-shot classification prompt is made of, and the baseline gains +20.50 against RESCUE's +5.50; over the other fifteen tasks it gains +0.71 against RESCUE's +1.46, and it is behind on nine of them.

The two also differ on the axis this paper is built around. Over the same sixteen cells the worst the baseline costs is −0.86, on TriviaQA, against RESCUE's −0.22; neither loses a full point anywhere, but the baseline's tail is four times deeper, and it improves ten of the sixteen tasks against RESCUE's thirteen. A cheaper probe buys a larger average here and a wider spread of outcomes, which is the trade the selector exists to avoid.

What this separates is where the probe's own attention is already the answer from where it is not. Where the continuation attends to spans the prompt makes explicit — labels in a classification prompt — one attention row suffices and the scorer adds little. Where the future-critical entries are not the ones the first generated token looks at, the row is a poor proxy and the learned residual is what recovers them. Both figures are on record: the pooled mean, and the per-task split that shows one task driving it.

The probe weight. One attention row against a 32-row observation window is otherwise outvoted, so the row enters the base policy's score with a weight; the runs above use unit weight. Sweeping it to 8 and 32 reaches +3.14 and +3.54 on the 16-task mean. Those weights are chosen on the evaluation set itself, which is why unit weight — the untuned setting — is the one reported here and in Section 5.4; the environment variable that sets it is exposed in the released code, so either number can be reproduced. The comparison that does not depend on the choice is the per-task one. At every weight the advantage is carried by TREC, and with TREC removed the ordering at unit weight reverses (+0.71 against RESCUE's +1.46) while even at weight 32 the margin falls to +2.08 against +1.46.

Two things are worth separating in Table 18. The first is that the selector is not the weak part. Its choices track the per-task ordering of the two policies closely — ForesightKV on the summarization tasks it actually wins, SnapKV on the extractive ones — and per document it edges out the per-task hindsight oracle over the same two policies (+0.50 against +0.44), which no fixed assignment of policies to tasks can beat. On GovReport and MultiNews, where ForesightKV is genuinely better,

CacheSel captures most of the difference ($+2.84$ and $+2.50$) while losing almost nothing on the eleven tasks where ForesightKV is worse.

The second is that this is not enough. CacheSel averages $+0.50$ and is barely better than even across tasks (8 wins, 6 losses, 2 ties), because both of its candidates are fixed policies and neither contains the entries the other one missed. The residual scorer produces a third cache that neither policy can produce — on Qasper it is worth $+7.31$ where the best available selection over the two policies is -0.89 — and the selector then gates *that*. What the selector contributes is verification; what the scorer contributes is something worth verifying.

N.2 When the Selector's Signal Is Reliable

Section 5.5 reports that the selector picks the retrospectively better cache on 55.6% of the documents where the two arms differ, and that its KL margin correlates with the realized score change at only $\rho = +0.11$. Both are averages over a quantity whose scale varies by four orders of magnitude, so they describe the selector where it has no signal as much as where it has one.

Splitting the deciding documents by $|\mathrm{KL}(\lambda{=}0) - \mathrm{KL}(\lambda{=}1)|$ separates the two regimes.

Table 19: Selector behaviour by the size of its own margin, over the $9{,}158$ deciding documents that carry a logged margin, in equal fifths. Accuracy is agreement with hindsight; the gain is realized points per document, which is zero on a rejection by construction. The margins are read from the runs with gate logging enabled; the abstention floor changes the decision on 24 of the $19{,}562$ logged probes (0.12%), so the quintiles are the same either way.

Margin quintile	**Range**	**Accuracy**	**Accepts**	**Gain / doc**
Q1 (smallest)	$< 4.8 \times 10^{-3}$	52.3%	50.8%	$+2.06$
Q2	4.8×10^{-3}–3.4×10^{-2}	53.7%	48.7%	-0.08
Q3	3.4×10^{-2}–1.1×10^{-1}	52.6%	48.2%	$+0.62$
Q4	1.1×10^{-1}–3.5×10^{-1}	55.3%	45.2%	$+1.20$
Q5 (largest)	$> 3.5 \times 10^{-1}$	$\mathbf{63.7}\%$	57.3%	$\mathbf{+5.91}$
All	—	55.5%	50.1%	$+1.95$

In Table 19 the top fifth is where the selector both decides well and decides on something worth deciding: 63.7% accuracy against 52.3–55.3% elsewhere, and $+5.91$ points per document against -0.08 to $+2.06$. It also accepts more often there (57.3%), which is the behaviour the mechanism predicts — a large margin means the corrected cache tracked the dense distribution visibly better, and that is the case where the correction is worth taking.

This is what the pooled 55.6% obscures. The selector is not a weak classifier applied uniformly; it is a classifier that is close to chance where its input carries no information and meaningfully better than chance where it does, and the documents in the latter regime are the ones with large score differences at stake. It also bounds what a better selector could buy: even perfect decisions on Q1–Q4 would move the average by less than the top quintile already contributes.

N.3 Selector Decision Outcomes

Section 5.5 reports that the one-token selector picks the retrospectively better cache on only 55.6% of the documents where the two arms differ, yet recovers 28.4% of the oracle headroom. Accuracy alone cannot explain that, because a decision's value is the size of the score difference it captures or forfeits, not whether it was right. Scoring every document under both arms with its task's own LongBench metric and keeping the $9{,}252$ on which the arms differ ($9{,}426$ are ties, and 72 further documents are excluded because their selector score matches neither arm), the decisions split as follows. The rate below is the same 55.6%, recomputed on this set.

Two asymmetries in Table 20 carry the result. The selector only acts when it accepts, so a wrong rejection forfeits an available gain but never causes a loss; and among accepts, the decisions it gets right are larger than the ones it gets wrong (16.86 against 11.47, a ratio of 1.47). Holding the hit rate fixed and equalising the magnitudes would leave $+0.70$ points per decisive document, against

Table 20: Selector decisions on the 9,252 documents where the base and corrected caches score differently. Rejecting returns the base cache, so rejections contribute nothing to the gain over the base policy — a wrong rejection forfeits the difference rather than losing it. Mean $|\Delta|$ is the average absolute score difference between the two arms in that outcome.

Decision	n	**Share**	**Mean $\lvert\Delta\rvert$**	**Contribution**
Accept, correction better *(right)*	2555	27.6%	16.86	+4.66
Accept, base better *(wrong)*	2109	22.8%	11.47	−2.61
Reject, base better *(right)*	2593	28.0%	15.17	0.00
Reject, correction better *(wrong)*	1995	21.6%	11.82	0.00
Net	9252	55.6% right	—	+2.04

Table 21: Documents moved by at least τ points, over all 18,750 evaluated documents, against each document's own base policy. "Ratio" is wins over losses: 1.0 is a coin flip.

	Correction only (λ=1)			RESCUE (gated)		
τ	Losses	Wins	Ratio	Losses	Wins	Ratio
1	23.4%	22.6%	0.97	10.4%	12.7%	1.22
5	14.5%	14.1%	0.97	6.0%	8.3%	1.39
10	9.2%	9.4%	1.02	3.7%	5.7%	1.55
20	5.1%	5.5%	1.07	1.9%	3.6%	1.87

the realized +2.04: about two thirds of the value comes from the magnitude asymmetry. Scaled back over all 18,750 evaluation documents this predicts +1.01 points for the selector over its base policies, which matches the +1.00 measured directly. That +1.00 is the document-weighted mean; the headline +1.28 is the unweighted mean over the 80 (base, task) cells, which is how LongBench reports and which gives the smaller tasks equal weight.

The tail, per document. Table 21 counts documents rather than cells, which is the level at which a user meets the method. The unconditional correction's losses and wins are the same size and the same number; the selector removes more of the losses than of the wins at every threshold, and the gap widens with the threshold.

Per base policy the acceptance rate tracks how much the base was missing, as the residual framing predicts: H2O, the weakest, accepts 62.9% of decisive documents and gains +4.23 per document; R-KV, the strongest, accepts 36.8% and gains +0.31.

The four rows above partition the decisive documents, so the value of a gate that keeps the selector's acceptance *rate* but chooses at random follows from them exactly. The mean of $s^{\text{corr}} - s^{\text{base}}$ over those 9,252 documents is +0.34, so accepting a uniformly random 50.4% of them returns +0.17 per document and accepting all of them returns +0.34. The selector returns +2.04. Its value is therefore not an artifact of how often it accepts — at a matched acceptance rate, choosing *which* documents to correct is worth twelve times choosing at random.

O Design Choice Sweeps

These are the three components of Appendix D that could have been tuned for accuracy. Each is swept below against its measured cost.

O.1 Probe Length

The default RESCUE configuration uses a single future probe token. Probe length is the one component of the method that could plausibly be tuned for accuracy, so we report it against its measured cost rather than selecting it on task scores. We vary the probe length over $\{1, 2, 4, 8\}$ on LaProx and SnapKV across all 16 tasks, and give the added logic cost alongside the most expensive predicted-future baseline we compare with, ForesightKV at 134.7 ms — which length 1 already exceeds.

Table 22: Probe length against added logic cost, over the 32 (base, task) cells on LaProx and SnapKV for which all four lengths completed. Gains are paired against each base policy's own score; p is a two-sided sign test against probe length 1.

Probe	Avg. gain	Worst drop	vs. p=1	Sign test	Added logic
1	$+1.26$	-1.50	–	–	**276.1 ms**
2	$+1.47$	-0.90	$+0.21$	0.248	330.4 ms
4	$+1.67$	-0.79	$+0.40$	0.002	442.9 ms
8	$+1.70$	-1.11	$+0.43$	0.001	682.6 ms

Latency is measured on SnapKV under the protocol of Table 14 (Qasper, 200 documents, median per document, exclusive GPU).

Longer probes genuinely help (Table 22): probe lengths 4 and 8 beat a single token by $+0.40$ and $+0.43$ points, and the improvement is consistent rather than marginal (21 of 32 cells improve at length 4 against 5 that regress, sign test $p = 0.002$; 22 against 4 at length 8, $p = 0.001$). The effect holds on both base policies (LaProx $+0.81 \rightarrow +1.31$, SnapKV $+1.71 \rightarrow +2.09$). We do not claim that one token is sufficient for accuracy.

We nevertheless ship probe length 1, for two reasons that are both about cost rather than accuracy. First, the added logic is already the highest of any method we compare against at length 1 — 276.1 ms against ForesightKV's 134.7 ms — so no setting on this grid buys back a cost advantage, and every step up widens the gap for very little: length 8 costs $2.5\times$ length 1 and returns $+0.43$ points. The cost is also not mostly the probe. Decomposing the 276.1 ms at length 1 — this sweep is its own measurement run, so it differs from Table 14's 277.2 ms by less than the run-to-run spread — the dense probe step itself is 20.6 ms and rebuilding and scoring the two candidate caches is 253.6 ms; the probe component scales linearly with length (20.6 $\rightarrow$ 158.6 ms from 1 to 8) while the trial component only doubles (253.6 $\rightarrow$ 515.2 ms), because constructing the two pruned caches is a fixed cost paid once per document. Second, the probe delays compression by exactly its own length — the dense prompt cache must be held for that many decode steps — and that delay is charged against the very memory argument the method rests on. The accuracy left on the table by this choice, $+0.43$ points, is the price of those two costs.

O.2 Where the Probe Is Read

The probe is the one part of the selector that must be *decoded*: a token is generated against the uncompressed prompt cache, which holds that cache for one extra step and is the 20.9 ms of Table 14. A variant avoids the decode entirely. The prefill pass has already produced logits at every prompt position, so the two candidates can be compared on the prompt's own trailing positions at no additional cost. We run that variant at 16 tail positions — sixteen times the evidence the shipped probe reads — on SnapKV/Qasper, LaProx/Qasper and LaProx/TREC.

It recovers little: $+0.00$, $+1.53$ and $+0.00$, against the decoded probe's $+7.31$, $+4.72$ and $+6.50$ on the same three cells. On two of the three it accepts no document at all and reproduces the base policy exactly.

The explanation is structural rather than a matter of tuning. The trailing prompt positions lie inside the observation window every base policy protects, so both candidate caches retain exactly the entries those positions attend to most, and the divergence the selector reads there is too small to separate them. The first position whose prediction the eviction decision can actually change is the one after the prompt, which is what the decode buys. A free probe is therefore not available at this budget, and the 20.9 ms is not the part of the cost worth attacking — the candidate caches, at 253.8 ms, are.

O.3 Number of Candidate Caches

The selector compares two candidates, $\lambda \in \{0, 1\}$ — the base cache and the fully corrected one. A finer grid would let it interpolate. We test $\lambda \in \{0, 0.5, 1, 2\}$ on SnapKV and R-KV over Qasper, MultiFieldQA-en and TREC, with the probe, features and checkpoint unchanged. The finer grid

Table 23: Restricting the correction to k contested slots, against the unrestricted correction, at a 128-token budget; SnapKV contests 96 of its 128 slots and R-KV 64, the rest being each policy's own protected recency window. Entries are gains over the corresponding base policy. Dashes mark cells not run.

Base	Task	Base score	k=16	k=32	k=64	Unrestricted
SnapKV	Qasper	25.15	+2.24	+3.88	+6.54	**+7.31**
SnapKV	TREC	47.00	–	+3.50	+4.50	**+5.50**
R-KV	Qasper	28.45	+1.90	+3.26	+4.29	**+4.80**
R-KV	TREC	60.50	–	+1.00	+0.00	+0.50

is worth $+0.07$ points on average over the six cells, and the sign is not consistent: SnapKV is flat (-0.05, $+0.03$, 0.00) and R-KV splits (-0.30 on Qasper, $+0.24$ on MultiFieldQA-en, $+0.50$ on TREC). The cost, by contrast, is not flat. From the decomposition above, building and scoring the candidate caches is 253.6 ms for two candidates and scales with their number, so four candidates costs roughly 507 ms of trial work — nearly four times ForesightKV's 134.7 ms, with no measured gain to show for it. We therefore keep two candidates, and note that this is a cost argument: we did not find evidence that intermediate λ helps, but six cells cannot rule it out either.

O.4 Restricting the Correction to k Slots

A natural way to make the correction safer would be to bound how much of the cache it may change: reserve the base policy's top-$(B - k)$ entries and let the corrected score compete only for the remaining k slots, so a bad correction can cost at most k entries. We implement this as a three-band score — reserved, contested, discarded — and sweep $k \in \{16, 32, 64\}$ against the unrestricted correction on SnapKV and R-KV over Qasper and TREC, with the selector left on.

The restriction does not pay (Table 23). On three of the four curves the gain rises monotonically with k and the unrestricted correction is best at every point — on SnapKV/Qasper, $+2.24$ at $k = 16$ against $+7.31$ unrestricted, so capping the correction at 16 slots discards two thirds of the available gain. The fourth curve, R-KV/TREC, is not monotone ($+1.00$, $+0.00$, $+0.50$), but it moves within a range narrower than a single TREC label and does not favor a small k either. The reason the restriction is unnecessary is that it duplicates work the selector already does, and does it worse: bounding the damage per document at k entries also bounds the benefit at k entries, whereas the selector rejects the correction entirely on the documents where it would hurt and leaves it unbounded on the rest. We report the sweep because bounding the blast radius is the obvious first thing to try, and it is worth recording that the measurement rejects it.

P Training Variance and the Learning Target

Every number elsewhere in this paper comes from one checkpoint per base policy, trained on 499 documents. To see how much of a reported gain is the recipe and how much is the draw, we retrain two base policies under two further seeds — H2O, where the correction helps most, and R-KV, where applying it unconditionally is harmful — changing the document split and the weight initialization and nothing else, and re-evaluate on three tasks spanning the range of outcomes.

Repeating the same three seeds for the *full-future* target answers the question the single-checkpoint comparison of Section 5.4 cannot. Table 24 gives both. The residual target is ahead on five of the six seed–policy pairs, by $+0.62$ on H2O and $+0.55$ on R-KV, and the gap is larger than either target's spread across seeds (0.18 to 0.49). The $+0.22$ reported over all 80 cells with one checkpoint each is therefore an underestimate of a difference these six pairs do separate, on three tasks and two policies rather than on the full grid.

Per cell the standard deviation across seeds is 0.58 on average and 1.15 at worst (Table 25). It barely shrinks when the three tasks are averaged — to 0.49 and 0.34 — because a seed change swaps the whole checkpoint, so all of a base policy's tasks move together rather than independently. That is the reason a margin cannot be made significant by adding cells alone.

Table 24: Gain over the base policy under three training seeds, for both learning targets, averaged over Qasper, TREC and LCC. Both targets use the same seeds, the same corpus split and the same selector.

Base	Target	Seed 0	Seed 1	Seed 2	Mean	s.d.
H2O (evict-once)	Residual	+2.91	+2.25	+3.45	**+2.87**	0.49
	Full-future	+2.08	+2.18	+2.50	+2.25	0.18
R-KV	Residual	+1.49	+0.84	+1.63	**+1.32**	0.34
	Full-future	+0.44	+0.96	+0.92	+0.77	0.24
Pooled					+2.10 vs +1.51	

Table 25: Gain over the base policy under three training seeds. The right-hand columns are the average over the three tasks, which is the quantity the paper's claims are built from.

		Gain by seed			3-task average	
Base	Task	0	1	2	s.d.	range
H2O	Qasper	+4.13	+3.56	+6.23		
	TREC	+4.00	+3.00	+4.50	0.49	1.19
	LCC	+0.61	+0.20	−0.39		
R-KV	Qasper	+4.80	+2.65	+4.38		
	TREC	+0.50	+0.50	+0.50	0.34	0.79
	LCC	−0.84	−0.62	+0.02		

Carried into the intervals, the seed term is a second source of error the task bootstrap does not see. Averaging the two policies' three-task spreads gives 0.415, so the standard error of a five-policy mean from this source is $0.415/\sqrt{5} = 0.186$; added in quadrature to the bootstrap's 0.428 it widens every interval by a factor of 1.09. The headline gain becomes $+1.28$ $[+0.45, +2.27]$ and still excludes zero. The budget contrast does not: $+0.40$ $[+0.07, +0.79]$ becomes $[-0.07, +0.95]$, which is why Section 5.4 reports it as a direction that holds on all five base policies rather than as a resolved interval. The $+0.22$ target margin does not exclude zero under either treatment.

Read against the paper's own effect sizes, this places them in three groups. RESCUE's overall gain of $+1.28$ is several times the spread and is safe. The budget contrast of $+0.40$ is comparable to it, but survives because its direction is the same on all five base policies and on 46 of 80 cells, which a seed-driven wobble would not be. The residual-versus-full-future margin of $+0.22$ is below it, and we do not claim it.

P.1 Residual Against Full-Future Supervision

This appendix expands the comparison between the residual target $\mathcal{K}_{\mathrm{GT}} \setminus \mathcal{K}_{\mathrm{base}}$ and the policy-agnostic full-future target $\mathcal{K}_{\mathrm{GT}}$ of Section 5.4.

One asymmetry is worth recording because it runs in a single direction. Counting cells where an arm loses at least a point against its base policy, the full-future variant is harmful on 7 and RESCUE on 3; every cell that separates them is one where the full-future target is harmful and the residual target is not (4 such cells, none in the reverse direction, McNemar $p = 0.125$). This is the behavior the residual framing predicts — supervision restricted to a policy's own misses should disturb what that policy already gets right less — but at four discordant cells it remains a tendency rather than a result.

On training cost the two are equivalent for a fixed deployment: a system commits to one base policy and trains one scorer either way, and the expensive part of the pipeline, the forward passes that capture the query and key/value states, is shared because the 18 features do not depend on the base policy. The full-future scorer's advantage is portability — the same weights transfer to a different base policy, which matters when several are served at once — and that advantage costs the small, consistent margin above.