REWIND: RESTORING EVICTED PREFIX STATES FOR HYBRID LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-context LLM agents repeatedly revisit shared histories during retries and branch exploration, making efficient prefix reuse essential for responsive serving. However, hybrid models cannot reconstruct an earlier recurrent state from the latest one, so cache eviction forces requests to recompute long prefixes even when they add only a short suffix. Optimizing checkpoint placement alone cannot recover the state once it has been discarded. Our key insight is that preserving complete boundary checkpoints can decouple prefix reuse from the lifetime of native cache entries. We present Rewind, a serving system that restores evicted prefixes for hybrid LLM agents. First, Rewind persists recurrent state, convolution history, and attention keys and values at a common token boundary, coordinating their installation across workers before execution resumes. Second, it uses cost-based checkpoint admission and retention to prioritize expected saved computation under storage pressure. Finally, its restore gate compares complete installation cost with the additional prefill avoided beyond the native cache hit. On a 16-session Qwen3.5-27B-FP8 replay using two RTX 4090 GPUs, Rewind's additional checkpoint store reduces prefill by 91.7%, replay duration by 30.5%, and return-round p95 latency by 26.5% relative to native caching. Under forced eviction at matched storage capacity, selective Rewind performs 41% less prefill than Snapshot-LRU, which saves every boundary and evicts by recency.

1 INTRODUCTION

Long-context agents repeatedly process shared histories as they continue a session, retry a step, or explore another branch (Luo et al., 2025; Zhu et al., 2026). A request may repeat tens of thousands of tokens while adding only a short suffix. Recomputing that history delays the next agent step and consumes serving capacity.

Prefix caching avoids this repeated work by retaining the model state associated with previously processed tokens (Kwon et al., 2023; Zheng et al., 2024). When a later request shares those tokens, the engine resumes from the cached boundary and prefills only the remaining suffix. Its benefit depends on both prefix overlap and the availability of the corresponding state.

Although prefix caching is effective when reusable states remain available, hybrid language models make recovery after eviction more demanding. They combine attention layers with recurrent layers that summarize preceding tokens in a state updated in place (Dao & Gu, 2024; Yang et al., 2025). The latest recurrent state cannot simply be sliced back to an earlier boundary. Continuing from that boundary requires its recurrent matrices, local convolution history, and the attention keys and values for the entire prefix. Saving attention KV alone is therefore insufficient: all components must describe the same tokens and position.

Existing systems have already explored prefix reuse in hybrid language models. Marconi jointly manages recurrent checkpoints and attention KV; its intermediate-node eviction releases that node's recurrent state (Pan et al., 2025). Sparse Prefix Caching optimizes checkpoint positions conditional on a retained cache entry (Shirokikh & Nikolenko, 2026). However, as contexts grow, their attention history occupies more cache space, increasing pressure to evict prefixes before they are reused. Once the native engine discards the required state, choosing a better checkpoint position cannot recover it. Reuse after eviction requires a complete boundary state that survives outside that cache.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

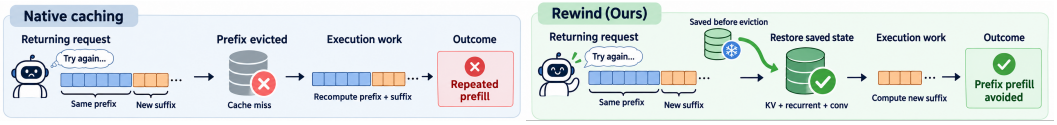

Figure 1: Native caching versus Rewind on a return (schematic). Left: intervening work has evicted the prefix's state, so the engine recomputes the repeated prefix. Right: Rewind saves a complete boundary state before eviction and restores it on return, prefilling only the new suffix.

To overcome this problem, we present Rewind, a serving extension for recovering evicted hybrid prefixes. Unlike native caching, whose reuse ends when the required state is discarded (Figure 1), Rewind persists complete boundary checkpoints in a separate store and reinstalls them when a later request reuses the prefix. It captures all state components at a common boundary, validates their identity and coverage, and coordinates installation across workers before execution advances. Existing inference kernels then process the remaining suffix. Rewind also manages the cost of persistence: periodic capture preserves reuse when the retained working set fits, while cost-based admission and eviction allocate limited storage to useful checkpoints. A restore gate compares complete installation cost with the additional prefill a record would avoid.

Our contributions are:

- We identify loss of complete boundary states as the main source of repeated prefill in our hybrid-agent replay: preserving states at native checkpoint positions captures 99.6% of the combined persistence-and-placement reduction.
- We introduce Rewind, which couples complete-state persistence with coordinated installation to recover evicted prefixes, and uses capture, retention, and restore decisions to control its storage and transfer costs.
- We show that Rewind's additional checkpoint store reduces prefill by 91.7%, replay time by 30.5%, and return-round p95 latency by 26.5% on the 27B replay. Under forced eviction at matched capacity, selective Rewind performs 41% less prefill than every-boundary Snapshot-LRU.

2 Motivation

2.1 Hybrid reuse requires a complete boundary state

Retaining a hybrid model's latest state does not make its earlier prefixes reusable. Prefill processes the input tokens before generation begins. For an attention layer, reusing a prefix of length O requires its key and value entries for tokens $[0, O)$. A recurrent layer instead applies an update $S_t = F(S_{t-1}, x_t)$ and retains the current state S_t. This update overwrites the preceding state: the presence of S_t does not provide S_O for $O < t$. Local convolution layers also need their recent input history. Even if attention KV survives, execution cannot resume at O without the matching recurrent state and convolution history.

A fixed-size recurrent state also does not imply a small checkpoint store. In our 4B configuration, recurrent matrices occupy 48 MiB per boundary, or 61.6 MiB with convolution histories. Keeping many boundaries multiplies this cost, and each complete checkpoint also requires attention KV through its saved offset. Unrestricted capture is therefore expensive even when individual recurrent states are compact.

2.2 Placement cannot recover a discarded checkpoint

Checkpoint placement chooses which token boundaries to save within a prefix. Given a reusable prefix of length L and a retained checkpoint at $O \leq L$, the engine restores at O and recomputes the $L - O$ intervening tokens. A checkpoint closer to L reduces this residual. Sparse Prefix Caching optimizes this choice for retained entries (Shirokikh & Nikolenko, 2026); Marconi addresses which entries survive through admission and eviction (Pan et al., 2025).

A less precisely placed checkpoint can nevertheless save far more work if it survives eviction. For a return to a 40K-token prefix, a surviving state at 39.6K leaves roughly 400 old tokens to process; a discarded checkpoint at exactly 40K saves nothing. In our resident-cache control, a 528-token stride

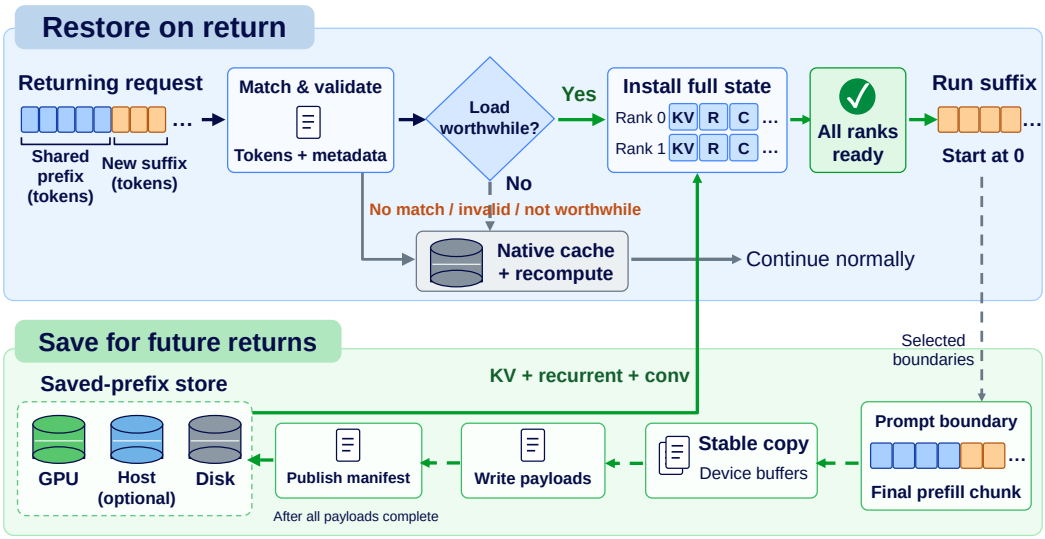

Figure 2: Rewind overview. Top: the engine matches and validates a record, plans the restored boundary, and installs the required state on each worker before the next forward pass. Records rejected during planning fall back to native caching and recomputation. Bottom: after the final prefill chunk, state is copied to stable buffers; the manifest is published only after payload writes complete. The saved-prefix store spans GPU, host, and disk.

leaves 782 uncached tokens per nominal 576-token append on average (Appendix E), far below the tens of thousands repeated after eviction. Section 4.2 measures these two sources of recomputation separately.

These observations motivate Rewind's two coupled operations: preserve the recurrent state, convolution history, and attention KV at a common boundary independently of native eviction, then reinstall them before the request proceeds from that boundary. Persistence preserves the starting point; coordinated installation makes it usable by the running engine.

3 Rewind

Rewind recovers evicted prefixes through four components (Figure 2): a writer persists complete boundary states (Section 3.1), an installer restores them before execution resumes (Section 3.2), a capture and retention policy allocates storage (Section 3.3), and a restore gate compares loading cost with saved prefill (Section 3.4).

Rewind adds a store of complete prefix states beside the native cache. The engine first uses its native cache, then considers a saved record if it extends that hit. An admitted record supplies the starting state for suffix prefill; otherwise prefill starts at the native boundary.

3.1 Persist a complete executable checkpoint

The checkpoint writer preserves all state needed to resume a prefix and exposes a record only after its payload is complete. A saved record pairs a prefix length O and token-prefix hash with recurrent matrices, convolution histories, and attention KV through O. Its metadata identifies the model configuration, dtype, kernels, KV block size and layout, and session version. Matching tokens finds a candidate record; matching execution metadata determines whether the current engine can use it.

Rewind captures state after the final prefill chunk of a prompt. At that moment, the recurrent matrices and convolution histories still describe the intended boundary. It first copies them into stable device buffers so later execution cannot overwrite a capture in progress. The writer then stages the complete payload, including attention KV, to storage. It publishes the manifest through an atomic rename only after every payload is complete. A partially written record is consequently never offered as a usable checkpoint.

3.2 REINSTALL STATE BEFORE EXECUTION ADVANCES

The installer turns a stored checkpoint into an executable prefix by restoring every state component before the scheduler advances past its boundary. The scheduler first validates a candidate's identity and coverage, plans the extra cached tokens, and allocates destination blocks. Each worker then loads recurrent matrices, convolution histories, and attention KV in its request-update hook, before the next forward pass can consume them. The planned computed-token offset becomes usable only with this installation. An incompatible or incomplete candidate is rejected during planning and uses normal prefill; a failure after admission aborts execution rather than continuing with partially installed state.

Requests in the same batch can restore different prefix lengths, including boundaries between native checkpoint positions. Rewind sets each request's offsets so that its next prefill starts at the restored boundary; existing model kernels handle the remaining tokens.

3.3 DECIDE WHICH CHECKPOINTS TO RETAIN

The capture and retention policy balances the cost of saving states against their expected reuse. Periodic capture controls the residual suffix when retained states fit; admission and eviction select among competing records when storage is scarce.

Capture frequency when retained states fit. With capture cost C, per-token prefill cost u, and returns at rate λ per processed token to uniformly distributed targets, the renewal checkpoint model gives $T^* = \sqrt{2C/(\lambda u)}$ (Young, 1974). Doubling or halving this interval raises modeled cost by only 25%, so the model motivates a coarse periodic schedule rather than precise checkpoint placement. Appendix B gives the assumptions and derivation; the deployed schedule applies the interval at prompt ends and adapts it after reuse observations.

Admission when storage is scarce. A candidate that would save S tokens upon reuse has expected benefit $p\,uS$, where p is the probability of token-identical prefix reuse during the period for which the checkpoint is retained. If c_0 represents capture and expected recovery overhead, the independent break-even decision is to save when

$$p \geq p^* = \frac{c_0}{uS}. \tag{1}$$

Unlike spacing, this threshold changes directly with the compute price. Underestimating u by a factor of two doubles the probability required to save a boundary. It can reject useful states even if the return predictor ranks them correctly.

Absolute probabilities near this cost-sensitive threshold matter even when candidates are ranked correctly (Elkan, 2001). Appendices B and C analyze probability error and compare predictors.

The deployed seven-feature logistic model estimates a return candidate within eight rounds from context and return history available at capture time. Its labels, derived from engine cache counters, estimate candidate returns and serve as a proxy for the analytical reuse probability p. For a candidate S tokens beyond the last capture, the implementation tests $\hat{p}uS \geq mC$, where the admission margin m multiplies estimated capture cost C. First-capture and maximum-gap rules ensure coverage. When storage exceeds its budget, Rewind evicts unprotected records with the lowest reuse-weighted saved computation per byte. The threshold describes one independent admission decision; eviction and coverage guards are heuristics for the shared budget. Appendix A specifies the guards and retention score.

3.4 SKIP RESTORATION WHEN LOADING COSTS MORE THAN RECOMPUTATION

The restore gate admits a checkpoint when its additional prefill savings justify installation cost. A small extension of the native hit may not repay loading. The gate therefore accounts for both the saved token count and the record's storage tier: GPU hot pool, host memory, or disk.

By default, Rewind restores any valid record that extends the native cache hit. A conservative alternative requires at least 528 additional reusable tokens. For costly transfers, a load gate compares estimated saved computation with installation time:

$$N_{\text{gap}}\hat{u}(1 + \beta q) > \alpha(\hat{c}_k + \hat{s}_k O)h(a). \tag{2}$$

Here N_{gap} is the extra number of tokens skipped and k is the storage tier. The fixed cost $\hat{c}_k$ covers boundary state and setup; $\hat{s}_k O$ accounts for the growing attention history. Queue depth q increases the value of avoiding prefill, while $h(a)$ discourages repeatedly loading a recent record. Appendix A specifies the settings.

The restore gate must estimate end-to-end installation cost. Copying the 48 MiB recurrent bundle takes 0.11 ms on-device, 3.4 ms from pinned host memory, and 41.3 ms in a disk-read microbenchmark. Complete installation also transfers attention KV, allocates blocks, and coordinates with the engine; its median is 388 ms across 233 installation-bearing requests in the 4B placement replay, whose median cached prefix is 32,456 tokens. Rewind batches contiguous attention spans per layer before scattering them into destination blocks. All storage-policy comparisons use this installer. Section 4.4 also tests whether disk restoration is worthwhile when the prefix stays cached and only a short residual can be saved.

Implementation. The prototype modifies approximately 1K lines of vLLM 0.25.1, uses its existing FLA kernels (Yang & Zhang, 2024), and adds a sidecar of approximately 4.1K lines to manage records and transfers. It supports content-addressed payload sharing, prefetch, and temporary leases; Appendix A gives their settings. Model files at a configured path must remain unchanged; replacing them requires invalidating the store. Captures are written asynchronously in the main replay and synchronously in the selective forced-eviction configuration. Our deep-context experiments use at most three active requests to fit the single-node memory budget.

4 Evaluation

We evaluate how much repeated prefill persistence removes beyond placement alone (Section 4.2), why selective retention helps at fixed capacity (Section 4.3), and when restoration costs less than recomputation (Section 4.4).

4.1 Experimental setup

Trace-driven replay. We combine TraceLab lengths and timing with public agent text in controlled serving replays (Zhu et al., 2026; Nebius, 2025; Exgentic team, 2026). Of 8,058 TraceLab sessions, 60.8% contain candidate returns identified from lengths and cache counters. Replay construction establishes exact reuse: a *declared return* matches an earlier token prefix, and ordinary appends retain preceding tokens. Appendix A gives the construction and a separate runtime study.

Models and workloads. We run Qwen3.5-4B on one 24 GB RTX 4090 and Qwen3.5-27B-FP8 on two GPUs with tensor parallelism (Qwen Team, 2026a;b). Each main replay has 16 sessions: six with many candidate returns and ten sampled from sessions with at least one candidate return. The 4B and 27B streams contain 731 and 611 rounds, respectively. Think time is capped at 45 seconds for 4B and 30 seconds for 27B; 27B contexts are capped at 60K tokens.

Both arms enable native prefix caching with three active requests. Rewind adds a 1 GiB hot pool per GPU and 64 GiB of host memory. The vLLM `gpu_memory_utilization` setting is 0.88 for both 27B arms; the 4B settings are 0.73 for native caching and approximately 0.77 for Rewind (exact configurations in Appendix A). They use the interval-based capture policy of Section 3.3 and restore whenever a saved record extends the native hit.

Policies at matched storage. A separate 27B workload stresses retention with 24 sessions, 1,133 rounds, and 93 declared returns to 40–55K-token contexts. Each return follows verified native eviction, and the estimated retained-state footprint is approximately 198 GiB (unique tokens times the driver's bytes-per-token assumption; Appendix A). Snapshot-LRU, a baseline we construct, saves every boundary and evicts by recency. Selective Rewind uses margin $m = 0.4$ in the capture-cost test of Section 3.3, eviction by reuse-weighted saved computation per byte, prefetch, and leases that temporarily protect a record for 120 seconds after restoration. It writes captures synchronously; Snapshot-LRU and the Marconi-style arm write asynchronously. A third arm adapts Marconi-style admission (Pan et al., 2025) within our persistence layer, rather than reproducing the published cache. All share a 64 GiB disk budget, a 32 GiB host tier, a 1 GiB hot pool per GPU, and the same installer.

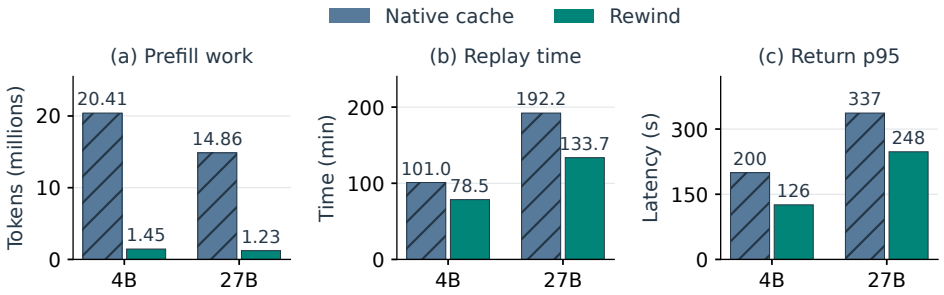

Figure 3: Saved prefixes reduce repeated work and the latency of returning requests. Matched-request 4B and 27B replays, with additional storage for Rewind: total prefill, replay duration, and return-round p95 latency.

Measurements. Prefill work is input tokens minus engine-reported cached tokens. Replay duration includes think time and capture completion. Return-round latency spans submission through the last output token, including queueing, restoration, prefill, and decoding; return-event time to first token (TTFT) stops at the first output token. Comparisons match request identities, lengths, and return labels, with no reported request errors. Appendix A gives records, repeatability controls, and statistical conventions.

4.2 Persistence accounts for most savings over native caching

Rewind reduces repeated prefill by preserving a starting state across native eviction (Figure 3). At 4B, work falls from 20.41 M to 1.45 M tokens and replay duration from 101.0 to 78.5 minutes. At 27B, work falls from 14.86 M to 1.23 M tokens and duration from 192.2 to 133.7 minutes. The latter removes 91.7% of prefill and 30.5% of total time. Its prefill ratio is 12.1×, with a paired-session bootstrap interval of 10.1–14.3× describing sensitivity to session mix. Native caching must rebuild a lost prefix; Rewind's complete-state installer lets the same kernels begin at the saved boundary. These results measure the benefit of the additional checkpoint store, with retention policies compared at equal capacity in Section 4.3.

Returning requests benefit directly. Their p95 round latency falls from 199.6 to 125.6 seconds at 4B and from 336.8 to 247.7 seconds at 27B. All 16 observed 27B sessions improve in prefill and mean round latency. A return can replace a long prefill with installation and a short suffix, but still incurs decoding and queueing; capture and transfer also consume time, and replay duration includes think time. The smaller reduction in elapsed time therefore reflects the portion of serving work that prefix restoration can remove.

The placement control distinguishes preserving a checkpoint from moving it closer to the return point. Saving complete states at native checkpoint positions reduces the 4B workload to 1.56 M prefill tokens; prompt-end capture brings it to 1.48 M. Persistence therefore accounts for 99.6% of the combined token reduction. The native-position checkpoint already skips most of a long prefix once it survives eviction; moving it to the prompt end removes only the residual between those boundaries. This explains why finer placement adds little in this replay, while remaining useful for partial overlaps within a retained entry. We compare token counts because the grid run shared its host.

Stream-construction controls. Three same-protocol controls probe how the replay's construction affects these numbers. Keeping only 13 of the 57 declared returns, with the rest converted to standard 576-token appends, leaves Rewind at 1.46 M prefill tokens against 21.46 M for native caching (run D25). Persistence therefore helps ordinary appends as well as explicit returns: either can reuse a prefix whose native state has been evicted. Rewriting 64 tokens in the tail of every return target costs Rewind 0.02 M additional prefill (1.47 M), as restoration falls back to the deepest clean anchor and prefills only the edited suffix. Matching the native arm's memory-utilization setting to Rewind's (approximately 0.77) changes its prefill by under 1% (20.49 M). The extra engine-memory allocation therefore explains little of this token gap. Appendix A records the protocols.

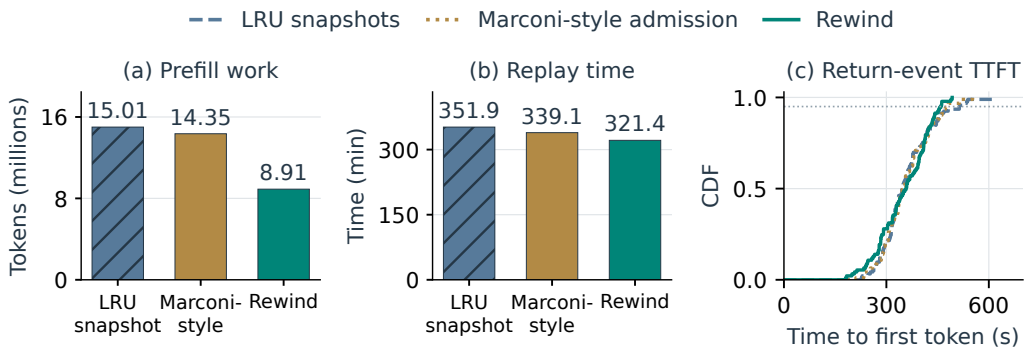

Figure 4: Selective Rewind performs less prefill than Snapshot-LRU at matched capacity. The three complete policy configurations share the storage hierarchy and budget on the 1,133-round forced-eviction stream. Marconi-style denotes our admission-policy adaptation. The TTFT distributions contain all 93 declared return events per arm.

4.3 A SELECTIVE CONFIGURATION IMPROVES REUSE AT FIXED CAPACITY

The matched-capacity experiment compares retention policies on a common complete-state store. Snapshot-LRU captures every boundary, while selective Rewind allocates captures and retention according to expected saved computation. The Marconi-style admission adaptation provides a third policy under the same storage hierarchy and installer.

Under forced eviction, native automatic prefix caching (APC) performs 38.53 M prefill tokens, close to the 38.62 M required without caching. Snapshot-LRU lowers this to 15.01 M by retaining complete states beyond native eviction. At the same storage budget, Rewind lowers it further to 8.91 M, a 41% reduction. Replay duration falls from 351.9 to 321.4 minutes (8.7%), and return-event p95 TTFT from 514 to 460 seconds (Figure 4).

Persistence explains Snapshot-LRU's gain over native APC; retention determines how much further Rewind can reduce prefill. Under the approximately 198 GiB estimated working set, capturing every boundary cannot keep every record in the shared 64 GiB disk budget. Recent captures compete with older prefixes that may avoid much more computation on return. Rewind's admission test limits low-benefit captures, its eviction score accounts for saved computation per byte, and leases protect recently restored records from immediate displacement. Across all rounds, Rewind starts 11% without a usable prefix, compared with 32% for Snapshot-LRU. This reduction in missed reuse is consistent with the retention mechanism.

The Marconi-style configuration performs 14.35 M prefill tokens in 339.1 minutes; Rewind uses 38% less prefill. Since these policies share the complete-state store and installer, this gap concerns which reusable states remain available, rather than whether hybrid state can be restored at all. The gains belong to the combined admission, eviction, prefetch, lease, and write-scheduling configurations; this comparison does not isolate the contribution of each component or reproduce the published Marconi system.

The timing differences support avoided prefill as the main explanation for the speedup. The recompute–Snapshot-LRU difference gives 0.28 ms of replay time saved per prefill token avoided; Snapshot-LRU–Rewind gives 0.30 ms/token. Applying the former rate to the latter token gap predicts about 28.5 of the observed 30.5 minutes saved. These end-to-end ratios include all configuration overheads, but their agreement indicates that preserving more usable prefixes explains most of the elapsed-time improvement.

Selectivity helps when records compete for storage, but can discard affordable reuse when they fit. In the main 4B workload, periodic capture needs 1.45 M prefill tokens and 78.5 minutes, compared with 3.59 M and 85.1 minutes for selective capture at the engine-default margin. Without capacity pressure, rejecting a checkpoint sacrifices a possible starting point without needing to make room for another record. This explains the use of periodic capture in the main replay and selective retention in the forced-eviction experiment. Appendix C examines admission-margin sensitivity under capacity pressure.

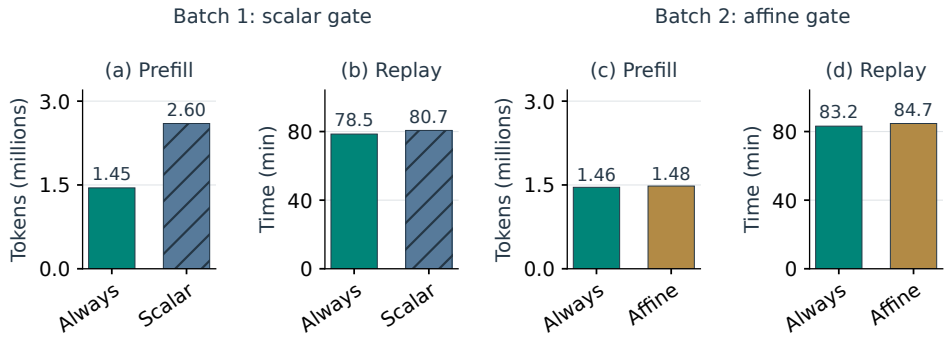

Figure 5: Restore gates on the 731-request 4B replay. Each experimental batch has its own always-restore control: (a,b) the scalar gate; (c,d) the length-aware gate (G-AFF) and its same-day control (G-AA). Corresponding metrics share vertical scales; small timing differences are within observed variation.

Benefits across architectures. Rewind also improves serving on Ring and Kimi. On Ring-mini-linear-2.0, it reduces prefill by 43% relative to native APC and completes the replay in 86.9 minutes, ahead of native APC (95.0) and Snapshot-LRU (90.9). On Kimi-Linear-48B-A3B-FP8, it reduces prefill by 85% relative to native APC; under a tighter 32 GiB host-plus-disk budget, calibrated Rewind performs 41.5% less prefill than Snapshot-LRU with comparable replay time. Appendix D gives the configurations and results.

4.4 Restore economics depend on complete installation cost

Restoration pays only when the extra prefill it avoids exceeds installation cost. For resident prefixes, median session time is 11.76 seconds with native caching, 11.43 with hot restoration, and 17.96 with disk snapshots. Here, native caching already skips most of the prefix. Restoring a later boundary removes only a short residual while still paying to install a complete state, including its attention history. The resulting disk slowdown explains why token overlap alone is insufficient for a restore decision.

On longer returns, conservative gating can instead discard useful reuse. On the 4B stream, requiring a 528-token restore gap performs almost like always-restore (1.47 versus 1.45 M prefill tokens). A gate with a single load-cost estimate rejects more reusable state (2.60 M), taking 80.7 minutes against its 78.5-minute always-restore control. The length-aware gate in Equation 2 recovers most of the lost work (1.48 M; run G-AFF), but takes 84.7 minutes versus 83.2 for the same-day always-restore run G-AA (1.46 M tokens; Figure 5). This small timing difference falls within observed variation: more detailed gating recovers reuse lost by the scalar estimate without demonstrating a speedup over always-restore on this workload.

The cost model separates two lengths that can differ substantially: loading scales with the entire saved prefix, whereas the benefit scales with its extension beyond the native hit. In a separate 27B fit, host-tier worker loading grows by 61 μs per prefix token ($R^2 = 0.88$), compared with the 280–300 μs per avoided prefill token inferred above. These measurements explain why a long missing prefix can repay transfer, while a long record extending an almost complete native hit may not. Fixed installation costs further penalize small gaps. Appendix E separates this fit from the 4B installation median.

The cold-read control further shows why the gate must use complete installation cost. Evicting payloads from the operating-system page cache makes attention-file reads 10.5$\times$ slower, yet warm and loader-level cold replays take 81.9 and 80.3 minutes. File-read latency is only one part of the path; allocation, transfers, and worker coordination also contribute, alongside the remaining model computation. A bandwidth microbenchmark alone therefore cannot determine whether restoration improves serving time.

5 Related Work

Hybrid prefix caching. Marconi stores recurrent checkpoints and attention KV in a radix tree and manages nodes using reuse opportunities and compute savings per byte (Pan et al., 2025). Its intermediate-node eviction releases the recurrent checkpoint while retaining KV in the child node. Rewind keeps a separate complete boundary record that can reinstall the released state. Sparse Prefix Caching stores sparse exact recurrent checkpoints with attention KV and resumes from the deepest available checkpoint (Shirokikh & Nikolenko, 2026). Its placement objective is conditional on retained entries; its prototype uses a last-K entry policy and CPU memory. Rewind evaluates persistence independently of placement and integrates complete-state reinstallation after native eviction. Placement still determines the suffix that remains.

Reuse by composition or reconstruction. HYPIC caches segment transitions, end states, and KV; composing segments at new positions requires seam-window recomputation to repair cross-segment attention (Liu et al., 2026a). LinearKV initializes from a cached chunk state and selectively recomputes positions to repair the combined context (Liu et al., 2026b). Both gain position-independent reuse while spending computation on correction. Tail-Replay omits recurrent checkpoints and rebuilds an approximate state from a short suffix of cached attention-output hidden states, overlapping replay with KV transfer (Liu et al., 2026c). Its replay budget trades reconstruction work against state approximation. Rewind chooses token-identical, position-aligned reuse and reinstalls recorded state without a reconstruction approximation.

Storage and agent serving. CachedAttention, LMCache, and Mooncake preserve attention KV across storage tiers (Gao et al., 2024; Liu et al., 2025; Qin et al., 2025). Rewind adds the complete state and installation ordering required by hybrid execution. Autellix and Continuum complement recovery by scheduling related agent requests (Luo et al., 2025; Li et al., 2026). Classical checkpoint spacing and cost-sensitive decisions (Young, 1974; Elkan, 2001) provide the capture-frequency and admission models.

6 Discussion and Conclusion

Limitations. The evaluation targets serving cost on constructed, token-verified replays; production reuse prevalence and end-to-end agent task quality remain to be measured. Our single-node deep-context runs use at most three active requests; higher concurrency requires bounded restoration buffers and backpressure (Appendix E). Application-side effects lie outside model-state recovery.

Conclusion. Rewind makes evicted hybrid prefixes reusable through complete boundary records and coordinated installation. On the 27B replay, it reduces prefill by 91.7%, replay duration by 30.5%, and return-round p95 latency by 26.5%. The resulting serving mechanism extends prefix reuse beyond native cache residency and supports cost-aware retention under finite storage.

AI Use Statement

AI assistants were used for code debugging, LaTeX editing, and proofreading during the preparation of this paper. All ideas, experimental results, analyses, and claims were produced and verified by the authors, who take full responsibility for the content.

Reproducibility Statement

Appendix A gives workloads, configurations, run records, and reproduction paths. Appendix B derives the cost analysis; Appendix D reports results on Ring and Kimi architectures. The submission attachment packages the experiment drivers; every recorded run cited in the appendices is identified by a record set.

REFERENCES

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *International Conference on Machine Learning*, 2024. URL https://proceedings.mlr.press/v235/dao24a.html.

Charles Elkan. The foundations of cost-sensitive learning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 973–978, 2001.

Exgentic team. Exgentic agent llm traces v2: Chat-only execution traces for agent benchmarks, 2026. URL https://huggingface.co/datasets/Exgentic/agent-llm-traces-v2.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with CachedAttention. In *Proc. USENIX Annual Technical Conference (ATC)*, pp. 111–126, 2024. URL https://www.usenix.org/conference/atc24/presentation/gao-bin-cost.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 1321–1330, 2017.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, New York, NY, 2023. ACM. doi: 10.1145/3600006.3613165.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Huanchen Zhang, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn LLM agent scheduling with KV cache time-to-live. *arXiv preprint arXiv:2511.02230*, 2026. doi: 10.48550/arXiv.2511.02230. Latest revision v7, September 2026. Also in ICLR 2026 Workshop on Lifelong Agents.

Yifei Liu, Juntong Wu, Yang Liu, Junhao Hu, Minghao Li, Xiaoxu Chen, and Weihang Chen. HYPIC: Accelerating hybrid-attention LLM serving with position-independent caching. *arXiv preprint arXiv:2607.01299*, 2026a. doi: 10.48550/arXiv.2607.01299.

Yirui Liu, Ruoling Qi, Longwen Wang, Xuaner Wu, Jian Chen, Yuxin Jin, Jiawei Shao, and Xuelong Li. LinearKV: One cached state suffices for position-independent caching in hybrid LLMs. *arXiv preprint arXiv:2608.11231*, 2026b. doi: 10.48550/arXiv.2608.11231.

Yirui Liu, Ruoling Qi, Xuaner Wu, Penghang Liu, and Jian Chen. Tail-Replay: Escaping the curse of linear attention in prefix caching for hybrid LLMs. *arXiv preprint arXiv:2608.30310*, 2026c. doi: 10.48550/arXiv.2608.30310.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. LMCache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665*, 2025. doi: 10.48550/arXiv.2510.09665.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for LLM agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025. doi: 10.48550/arXiv.2502.13965.

Nebius. Swe-rebench openhands trajectories. https://huggingface.co/datasets/nebius/SWE-rebench-openhands-trajectories, 2025. 67K real OpenHands attempts over real-world Python repositories.

Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. Marconi: Prefix caching for the era of hybrid LLMs. In *Proc. Machine Learning and Systems (MLSys)*, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/7c180af017258d239bac6248d1eb26ac-Abstract-Conference.html.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a kvcache-centric architecture for serving llm chatbot. In *Proc. USENIX FAST*, pp. 155–170, 2025. URL https://www.usenix.org/conference/fast25/presentation/qin.

Qwen Team. Qwen3.5-4b. https://huggingface.co/Qwen/Qwen3.5-4B, 2026a.

Qwen Team. Qwen3.5-27b-fp8. https://huggingface.co/Qwen/Qwen3.5-27B-FP8, 2026b.

Mikhail Shirokikh and Sergey Nikolenko. Sparse prefix caching for hybrid and recurrent LLM serving. *arXiv preprint arXiv:2605.05219*, 2026. doi: 10.48550/arXiv.2605.05219.

Songlin Yang and Yu Zhang. flash-linear-attention: A library for efficient linear attention. https://github.com/fla-org/flash-linear-attention, 2024.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/4904fad153f6434a7bcf04465d4be2cc-Paper-Conference.pdf.

J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974. doi: 10.1145/361147.361115.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Tracelab: Characterizing coding agent workloads for llm serving. arXiv:2606.30560 and https://tracelab.cs.washington.edu, 2026. Sanitized real-world trace of Claude Code and Codex.

A WORKLOADS, SETTINGS, AND REPRODUCTION

Replay construction. TraceLab supplies lengths, timing, and cache counters. A boundary within $\max(64, 0.05T)$ tokens of the previous input length T is classified as an append, with 1% tolerance for resubmissions; other reductions identify candidate returns. These labels do not prove token identity. Replay inputs use public agent text, and declared returns explicitly match an earlier prefix. Main streams combine six heavy-return and ten sampled sessions with at least one candidate return; the 27B sample uses seed 66 and requires a deep-return candidate. Forced-eviction returns use disjoint branches, relative depths 0.20–0.56, and at least eight intervening rounds. All 93 events pass the recorded cross-branch collision check.

Instrumented revision operations. A separate runtime study uses Qwen3.5-4B with LangGraph coding/test-failure workflows, WideSearch search/verification tasks, and Exgentic v2 message-trace replay. Its 1,466 calls contain 48 rollbacks, 85 branch returns, 69 forks, and 270 same-step retries, totaling 472 revision operations. This category includes operations such as forks that need not revisit an earlier prefix; it is distinct from the replay's token-identical declared returns. The driver is `revision_runtime_study.py` in the code attachment, and the aggregate is record set R20.

Storage-footprint estimate. The forced-eviction trace has 3,060,836 unique tokens. Its driver assumes 69,632 bytes per token for the TP2 retained-state footprint, including fp16 attention KV and recurrent/manifest overhead: $3{,}060{,}836 \times 69{,}632/2^{30} = 198.50$ GiB. This is a linear estimate from the driver's 68 KiB/token assumption, not a census of all saved-boundary files. Sources are record set R19 (trace and collision-check files) and the driver `trace_construction.py` in the code attachment.

Statistics. Return p95 is sorted index $\lfloor 0.95n \rfloor$ (zero-based). The bootstrap resamples the 16 paired 27B sessions 10,000 times with seed zero, sums each arm's prefill, and takes their ratio. It measures session-mix sensitivity within the recorded runs, not run-to-run uncertainty. Each reported configuration uses one complete replay, supplemented by A/A repeatability controls. Same-day A/A reruns vary by approximately two minutes. The duplicated 27B native arm agrees to 0.5% in duration and 1.2% in prefill (192.2/193.1 minutes; 14.86/15.04 M tokens); small timing differences are therefore descriptive.

Code attachment and record sets. The submission attachment contains the experiment drivers under formal module names (`code/rewind/`); drivers resolve the workspace root at run time and read recorded runs under `results/`. Record sets R1–R10 and R16–R21 identify the recorded runs behind the experiments below (Table 1); we cite a record as $\texttt{R}n/\langle\texttt{file}\rangle$.

Stream-construction controls. The controls of Section 4.2 rewrite the 4B round plans before replay; round count, timing, and request identities are preserved, and the two arms replay byte-identical plans at each setting. *Dilution (D25 at 25%)*: exactly 29 (50%) or 13 (25%) of the 57 declared returns are kept, sampled per session with a seed fixed by the rate, and the remainder become 576-token appends. Native prefill is 20.62 M at 50% and 21.46 M at 25% (20.41 M undiluted); Rewind stays at 1.44–1.46 M, showing that appends also retain useful long-prefix reuse after native eviction. *Edited returns*: 64 tokens inside the final 96 of every return target are replaced; Rewind prefills 1.47 M (1.45 M clean), because the deepest anchor below the edit still applies and only the edited tail is recomputed — an edit earlier in the context costs proportionally more of the prefix. *Equal allocation*: native caching at `gpu_memory_utilization`=0.7717 performs 20.49 M tokens (+0.4%), and a Rewind rerun (H20) with a 20 GiB host tier performs 1.46 M (1.45 M at 64 GiB), so neither the engine-memory nor the host-tier setting drives the comparison.

Capture and restore settings. The periodic policy evaluates $T^* = \sqrt{2C/(\lambda u)}$ at each prompt end, clips it to [1,056, 16,384] tokens, and captures once the gap from the last capture reaches that threshold. The first boundary is captured unconditionally; a prompt end before the last captured offset resets the interval. The 4B initialization uses $C = 60$ ms, $u = 66\,\mu$s/token, and $\lambda = 5.4 \times 10^{-5}$, yielding about 5.8K tokens. A restore multiplies the rate estimate by 1.6, capped at eight times the prior; each capture decision decays it by 0.98 toward the prior. This adaptation temporarily shortens the capture interval after reuse.

The deployed logistic model is fitted on 1,947 eligible TraceLab histories, split 70/30 by session, to predict a candidate return within eight rounds (AUC 0.758). Its seven inputs encode depth below the longest context seen, rounds since a return, distinct prior return targets, a smoothed restore indicator, proximity to prior targets, context length, and time since a return. Online restore observations update the history with weight 0.3. The prediction target is a return candidate derived from engine cache counters; token identity remains unobserved in those labels. The 27B capture test uses margin $m = 0.4$; tighter-budget Kimi uses an offline-calibrated $m = 1.5$. In $\hat{p}uS \geq mC$, m is a multiplier on capture cost, not an additive probability offset. The first boundary is captured unconditionally; subsequent candidates normally require a span of at least 256 tokens, and a 12,288-token gap forces a capture.

Under storage pressure, the eviction weight is the maximum of the session's smoothed observed reuse rate and the logistic return-candidate score. For each eligible record, the implementation multiplies this weight by the incremental tokens saved over the preceding retained checkpoint and a fixed prefill-cost estimate, then divides by its own-file bytes. Shared payload pools count once toward the budget. The lowest-scoring eligible record is removed first. In-flight restores and the newest checkpoint of a session with multiple records are protected. A post-restore lease temporarily protects an installed record for 120 seconds, subject to the configured residency budget; prefetch can stage a host- or disk-served record into the GPU pool. Content addressing shares identical payloads. These features are part of the measured configuration, not separately attributed sources of speedup.

The length-aware restore gate updates tier estimates with decay 0.97. Initial costs are 25 ms + 15 μs/token for GPU, 50 ms + 20 μs/token for host, and 150 ms + 25 μs/token for disk. Prefill estimates are 66 μs/token at 4B and 330 at 27B. We use $\alpha = 1$, $\beta = 0.3$, and $h(a) = 1.25$ if the source/offset recurs among the source's last four admissions within 120 seconds, otherwise 1. Scalar controls start at 450 ms for GPU and 2,000 ms for host/disk, with update weight 0.3 and $\beta = 0.15$.

Table 1: Paper experiments and their record sets. The driver module for each set is included in the submission code attachment. All main figures use existing recorded runs.

Experiment	Set	Arms / records
Main 4B	R1	`nalign3`, `rwalways`
Main 27B	R2	{`nalign`,`rwlite`}`_seedA`
Placement	R3, R4	`gridspill`; `rw3`
Matched policies	R5, R6	main; Marconi-full
Restore / Kimi	R7, R8	gate; calibrated
Prediction / oracle	R9, R10	oracle; theory
Loading	R16	cold I/O
Stream controls	R17	dilution

Sets R18–R21 cover the resident native-cache control (Appendix E), the trace-construction checks above, the revision-runtime aggregates, and the pilot replay rounds (Appendix C).

Table 2: Distinct 4B control runs. Record paths are `set/file` with sets from Table 1; equal rounded totals do not denote the same run.

Run	Setting	Record	Prefill tokens
D25	13 declared returns	`R17/rw25.json`	1,461,937
G-AA	Always-restore, 57 returns	`R7/rwalways2.json`	1,457,769
G-AFF	Length-aware gate	`R7/rwaffine.json`	1,480,249
H20	20 GiB host tier	`R17/rw100.json`	1,457,409

Availability and declarations. The submission attachment contains the experiment drivers under formal module names (`code/rewind/`); analysis drivers read recorded runs directly, and serving drivers require the corresponding GPUs and model weights. Recorded runs are not bundled with the attachment; public artifact hosting and trace-use permissions remain to be finalized. No new human-subject responses were collected. Stored states require the retention permissions of their underlying application data. Author contributions, funding, and competing-interest declarations must be supplied by the authors for submission.

B Cost Analysis

Deriving the spacing rule. Assume $C, u, \lambda > 0$ and positive i.i.d. gaps X with finite second moment, independent of the return process. Captures are retained indefinitely. For a uniform target in a long covered prefix, a gap of length x is sampled in proportion to x; the mean residual inside it is $x/2$. The renewal limit therefore gives $\mathbb{E}[X^2]/(2\mathbb{E}[X])$ residual tokens. Capture frequency is $1/\mathbb{E}[X]$, giving

$$J = \frac{C}{\mathbb{E}[X]} + \frac{\lambda u\, \mathbb{E}[X^2]}{2\mathbb{E}[X]}. \tag{3}$$

With $\mu = \mathbb{E}[X]$, Jensen's inequality yields

$$J \geq C/\mu + \lambda u \mu/2 \geq \sqrt{2C\lambda u}.$$

Equality requires constant gaps of length $T^* = \sqrt{2C/(\lambda u)}$. Substituting $T = rT^*$ gives

$$\frac{J(T)}{J(T^*)} = \frac{1}{2}\left(r + \frac{1}{r}\right). \tag{4}$$

A k-fold error in either price or rate changes T^* by $\sqrt{k}$ and multiplies cost by $(\sqrt{k} + 1/\sqrt{k})/2$. At $k = 2.23$, this is an increase of approximately 8.2%. This model excludes eviction and transfer contention.

Bounding the admission error. Let $v = uS$ and $a \in \{0, 1\}$ denote admission. Expected incremental cost is $L(a) = a(c_0 - pv)$. Its minimizing decision admits at $p^* = c_0/v$. If p and $\hat{p}$ lie on the same side of p^*, the plug-in decision has zero excess cost. Otherwise its excess is $v|p - p^*| \leq v|p - \hat{p}|$. Averaging gives Equation 5. If $p^* > 1$, no probability can justify admission and excess is zero. Explicitly,

$$\mathbb{E}[\text{excess cost}] \leq \mathbb{E}[uS\,|\hat{p} - p|]\,. \tag{5}$$

The bound uses pointwise error relative to token-identical reuse probability. Aggregate expected calibration error (ECE) (Guo et al., 2017) need not bound it or determine the policy's cost. Training on candidate-return labels introduces a further proxy error; Equation 5 does not remove that discrepancy.

C Admission and Predictor Diagnostics

An offline oracle with future knowledge and unconstrained retention uses 3.11 M prefill tokens on the forced-eviction stream. Relative to recomputing 38.62 M tokens, Snapshot-LRU realizes 66.5% of the possible reduction and Rewind 83.7%. This is a reference for remaining reuse opportunities, not a competitor with the same information and capacity. Of Rewind's remaining 5.8 M-token gap, 41% comes from rejected captures, 38% from records evicted or not yet staged, and 21% from restoring a shallower boundary.

Admission is the largest of these terms, and its price matters in practice. Raising the 27B margin from 0.4 to 0.8 and 1.2 increases measured prefill from 8.91 M to 9.97 M and 10.67 M tokens. The simulation using the nominal prefill-cost estimate preferred 1.2, which costs about 20% more prefill in the engine than 0.4. This behavior is consistent with Equation 1: a mispriced capture can be rejected despite a useful probability estimate.

A separate offline predictor comparison examines ranking and probability calibration (Table 3). The MLP has higher area under the ROC curve (AUC) than the deployed logistic model, yet larger expected calibration error (ECE) and a smaller fraction of the available prefill savings. The twelve-feature logistic model slightly improves savings despite a higher ECE. These results favor evaluating predictions through their cache decisions; neither AUC nor aggregate ECE alone determines serving cost.

Predictor comparison. Table 3 uses retrospective labels derived from replay length reductions, with 528-token matching tolerance, on 1,133 main and 1,451 pilot rounds. Predictive metrics hold

Table 3: Offline predictor comparison. Oracle savings recovered is the fraction of the recompute-to-oracle prefill reduction realized in simulation. Predictive metrics use held-out sessions; the policy simulation interleaves all sessions. Appendix C specifies the labels and retrospective features.

Predictor	AUC	ECE	Oracle savings recovered (%)
Logistic, seven features (deployed)	0.851	0.044	71.0
Logistic, twelve features	0.914	0.055	71.9
MLP	0.980	0.128	57.6

out alternating sessions; ECE uses ten equal-width bins. The cache simulation interleaves all sessions, including training sessions. Expanded models include completed-session length in a trajectory feature, so they are offline diagnostics, not causal replacements for the deployed model. The percentage of oracle savings recovered is $100(P_0 - P)/(P_0 - P_{\text{oracle}})$, where P is simulated prefill and P_0 is recomputation work. These results show an association between prediction and cache decisions, not an isolated causal effect of calibration.

D Results on Ring and Kimi

Rewind's gains extend to both Ring-mini-linear-2.0 and Kimi-Linear-48B-A3B-FP8, with faster replay on Ring and substantial prefill savings on Kimi under constrained storage.

Ring. At a 64 GiB disk budget and 32 GiB host tier, Rewind completes the replay in 86.9 minutes, compared with 95.0 for native APC and 90.9 for Snapshot-LRU. Prefill is 5.46 / 2.90 / 3.13 M tokens for native APC / Snapshot-LRU / Rewind. Rewind thus removes 43% of native prefill and achieves the shortest replay time among these configurations, showing that total serving time also depends on capture and transfer overhead.

Kimi. With the same 64 GiB disk and 32 GiB host settings, the initial Rewind configuration reduces native prefill by 85% and replay time from 200.1 to 180.2 minutes. Native APC / Snapshot-LRU / Rewind use 24.85 / 3.04 / 3.84 M prefill tokens and take 200.1 / 174.3 / 180.2 minutes. Calibrating admission brings Rewind's replay time to 173.8 minutes (4.85 M prefill), comparable to Snapshot-LRU's 174.3 minutes. Its retention advantage becomes clear under a tighter 32 GiB host-plus-disk budget: calibrated Rewind reduces prefill from 8.14 M to 4.76 M, a 41.5% reduction, while completing the replay in 174.0 minutes versus 175.2 for Snapshot-LRU. These results illustrate how cost-aware admission preserves more useful computation as storage becomes constrained.

E Installation Cost and Concurrency

Residual and installation-cost measurements. The resident native-cache control `R18/nalign_inf.json` records 201 appends and 157,088 total uncached append tokens, averaging 781.53 per append. Subtracting the nominal 576-token append gives an excess of 205.53 tokens; some final appends are shorter, so this nominal excess is not an exact old-token count. The session-time comparison uses this control's `nalign_inf`, `rwlite_inf`, and `snap_inf` arms. Its reported session medians follow the driver's upper-middle order statistic for 16 sessions.

The 388 ms installation median comes from 233 requests with positive reported restore time in `R4/rw3.json`; their median cached prefix is 32,456 tokens. This pools storage sources and uses request-level summaries. The 27B host-loading fit instead uses 1,354 per-rank entries from `R5/rewind_loads.jsonl`, summarized in `R16/install.json`. Their median restored boundary is 29,761 tokens and median worker loading time is 1,568.5 ms; the fitted slope is 61.24 μs/token ($R^2 = 0.877$). These differ in model, workload, tier, and aggregation, so the slope and the 4B median are not interchangeable installation estimates.

Transfer and residency. Evicting real payloads before rereading makes attention-file reads 10.5× slower than warm reads; direct I/O gives 8.9×. In the end-to-end 4B control, warm and loader-level cold runs take 81.9 and 80.3 minutes, within observed timing variation.

Concurrency. A 421-request replay completes at active-request caps three and six; the simple GPU/disk configuration takes 48.1 and 32.4 minutes, respectively. Full deep-context runs can exhaust memory at cap six: one run terminates seven of 566 requests.