COARSE KV RESIDENCY: DECOUPLING KV REACHABILITY FROM GPU MEMORY IN LONG-CONTEXT RL ROLLOUTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic RL rollouts alternate generation with tool waits, during which each sequence's complete KV state must remain in GPU memory. In a replay of a production turn structure, 85.6% of resident KV belongs to sequences idle in such waits. However, this footprint cannot be reduced by eviction or approximation: rollouts are the training data of RL post-training, and perturbing the sparse top-k selection changes the policy being optimized. We propose *coarse KV residency*, which decouples logical KV reachability from physical residency: the host stores every sequence's complete KV state, while the GPU maintains a bounded pool per sequence, layer and KV group. Pool capacities are sized at admission from measured sparse-attention demand and maintained by warm-start initialization, LRU retention and periodic reallocation. Residency never changes the model's own top-k selection, and missing selected entries are fetched before attention reads them. On Qwen3.5-4B, pooled decoding is bit-exact against full residency and the selected configuration passes all sixteen registered noninferiority endpoints, while allocation remains within 2.5–13% of an oracle, admission seeding reduces K/V traffic by 25.7%, and periodic reallocation reduces replacements by 40.3% on long rollouts. The freed residency yields up to 88% higher aggregate decode throughput through additional concurrent rollouts.

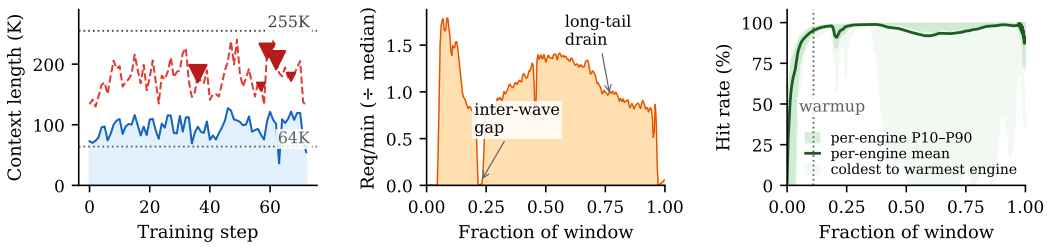
1 INTRODUCTION

RL post-training is one of the largest costs of building frontier models. Xiaomi live-streamed the MiMo-V2.6 RL run: about 750K trajectories in under six days, up to 2.6M dollars per run (Xiaomi MiMo Team, 2026). Much of this cost is rollout generation, and these runs are agentic: a trajectory alternates generation with tool and environment waits. During a wait the sequence produces no tokens, yet its complete KV state must survive until the trajectory ends. Holding that state in GPU memory blocks admissions. At 255K context, one full-resident sequence caps a two-GPU instance at batch 8 (Table 4). Replaying a production turn structure, 85.6% of resident KV sits in parked sequences, and a median of just 2 of 20 admitted sequences generate (Figure 16, §I).

Sparse attention serves such contexts cheaply, but RL holds it to a stricter standard than inference, which tolerates a slightly degraded answer. In RL the rollouts are the training data: selection errors enter their scores, and because rollouts are judged against each other in groups, one perturbed rollout distorts its whole group's evaluation. The price is measurable: scoring attention on a retained subset instead of all visible tokens shifts the logged action's log-probability by $+0.62$ nats on average, one-signed in 56 of 64 requests (§5.1). The top-k selection therefore stays exactly the model's own. Only residency may change. Offload is the natural lever, but a fully offloaded sequence must rebuild its GPU state when it resumes. The target is to keep sequences partially resident, so fixed hardware carries more concurrent rollouts and the wall-clock shortens.

We profile the indexer's sparse selection during decode, and the measurements supply the insights for KV offload in RL (§3). Most resident KV is idle at any moment, but idle entries return after delays no cache horizon can predict, so they must stay reachable rather than be dropped. How much KV a sequence needs varies by almost an order of magnitude across sequences and layers and is measurable at admission. Demand drifts fast from step to step but settles over thousands of steps,

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



(a) Rollout context length against the 64K residency budget and the 255K context limit.

(b) Serving request rate divided by its window median; load arrives in waves.

(c) Prefix-cache hit rate per engine: mean and P10–P90, coldest to warmest.

Figure 1: Workload of a production agentic-RL run: rollouts sit between the residency budget and the context limit, serving load arrives in waves, and prefix reuse is high on most engines but not all (serving axes window-relative).

so retention needs a fast and a slow mechanism. The CPU-to-GPU traffic this demand requires fits inside the idle window before each attention, as long as it moves in pool-sized transfers.

We propose *coarse KV residency* (§4). The host keeps every sequence's complete KV state. The GPU keeps a coarse pool per sequence, layer and KV group. Built on these insights, a pool-maintenance algorithm balances the KV payload in HBM against host-to-device PCIe traffic without touching the attention computation: the top-k selection stays the model's own, and the pool only decides what resides. Admission sizes each pool once from its measured demand, a warm start seeds it, LRU retention follows realized demand step by step, and periodic reallocation releases stale residency on the slow clock. An early indexer, decoupled from the critical path, scores ahead of the forward pass and opens a prefetch window in which each pool's missing entries arrive, in one transfer per pool, before that layer's attention begins. A late arrival is waited on, never replaced. Related work is collected in Appendix A.

We evaluate end to end on Qwen3.5-4B: five two-stage sparse branches (a same-step baseline, three early-indexer leads, and IndexShare (Z.ai, 2026; Bai et al., 2026)), each with coarse KV residency and host offload (§5). The three early-lead branches stay within 1.05 points of the baseline, and the selected branch clears the registered noninferiority gate on all sixteen endpoints. Allocation lands within 2.5 to 13% of an oracle, the seed cuts K/V traffic by 25.7%, and reallocation saves 40.3% of replacements. Where a lead window exists, prefetch hides fully under the pre-attention chain and the decoupled schedule models decode at $1.24\times$ the resident speed; the selected branch $d^*{=}0$ has none, and its case is the freed residency (§5.4).

Our contributions are:

1. We profile the indexer's sparse selection during decode and extract the insights for KV offload in RL: resident KV is mostly idle yet returns on unpredictable horizons, demand is heterogeneous and measurable at admission, drifts on two clocks, and moves in pool-sized transfers (§3).
2. We design a coarse-pool residency algorithm that lowers host-to-device KV transfer without changing the top-k selection (§2, §4).
3. We implement a system that decouples the indexer from the critical path, opening a prefetch window that removes indexer compute and KV transfer from the attention path (§4).
4. End-to-end evaluation on fully sparse-trained models shows nearly lossless accuracy and bit-exact pooled decode; where a lead window exists, the decoupled schedule models $1.24\times$ resident speed (§5).

2 Background and Problem Formulation

Native sparse attention in long-context RL.

Sparse attention is a low-cost way to serve long contexts: a lightweight indexer scores all visible KV entries, and attention reads only the top-k subset. The pattern is now built into pre-training or mid-training itself: DeepSeek (DeepSeek-AI, 2025; 2026b;a), GLM-5.3 (Z.ai, 2026) and MiniMax-

M3 (MiniMax, 2026) all train the indexer with the model. Models in this DSA (DeepSeek Sparse Attention) paradigm are the targets of this paper. In this setting, at sequence i, layer ℓ, KV group g (the unit that holds one index) and step t, the selection is

$$S_{i,\ell,g,t} = \arg\text{top-}k\big(s_{i,\ell,g,t}\big), \tag{1}$$

with the score s over all visible keys. The model adapts to this sparse representation during sparse training, and at inference the indexer scores the full visible history and selects the top-k on every step. Residency is a separate question: the resident set $R_{i,\ell,g,t}$ (§2) decides where entries live and when they arrive, but never affects the indexer's top-k selection.

Why RL rollouts turn KV into a residency problem.

An agentic rollout alternates generation with tool waits, and a parked sequence keeps its KV state (its next query still scores the full history) yet produces no tokens. The gap between the admitted batch and the token-producing batch is the *batch hole*, and it is the common case: replaying the production turn structure on our testbed, 85.6% of the resident KV is held by parked sequences, and at 255K full residency caps the batch at 8 (Table 4, §I). Figure 1 shows the workload: rollout length sits between the full-residency budget and the context limit, and prefix reuse is high but uneven across engines. The optimization target is useful active rollout concurrency per resident HBM, raised by decoupling how much KV a parked sequence keeps on the GPU from its right to remain in the batch.

Problem formulation.

Let $\mathcal{K}_{i,\ell,g}$ be the complete KV state of sequence i, layer ℓ, group g, in host memory, and $R_{i,\ell,g,t} \subseteq \mathcal{K}_{i,\ell,g}$ the entries resident on the GPU at step t. Each rank r carries a token-entry budget B_r, and correctness requires every selected entry to be resident before its layer's kernel reads it:

$$\sum_{(i,\ell,g)\in r} \big|R_{i,\ell,g,t}\big| \;\le\; B_r, \tag{2}$$

$$S_{i,\ell,g,t} \;\subseteq\; R_{i,\ell,g,t} \quad \text{at first read.} \tag{3}$$

The goal is to keep $|R|$ small so (2) admits many sequences, while minimizing the cost of (3): host-to-device traffic into R and stalls of late-covered reads. The schedule must act before S itself is computed, so Section 4 prefetches a prediction $\hat{S}$ of S. Every entry of S is then resident or in flight at first read. The schedule is tractable because a sequence's working set stays far smaller than its context and cold entries return after long absences, so $R \ll \mathcal{K}$ while its complement stays reachable. Section 3 measures both properties. Section 4 derives the schedule.

3 Characterizing KV Residency in Native Sparse Attention

We profile DeepSeek-V3.2-Exp (DeepSeek-AI, 2025) and observe how its attention selection distributes and evolves during decode, to ground the algorithm design. We sample the indexer of ten uniformly spaced layers on 54 natural-language 128k LongBench-v2 documents and store, in descending score, the top-64,000 key indices at each of the last 20,000 consecutive query positions. For every sequence and layer, we examine how each query's top-2048 selection changes and how it sits inside the top-64,000 ranking, and analyze the results along two axes: layer heterogeneity and temporal behavior. Each (sequence, layer) pair is one pool, 540 in total. The model is MLA-based, so KV-group heterogeneity does not enter the discussion.

Observation 1: Most KV is cold, but cold KV returns. The dead-mass ledger (Figure 2) counts each entry's selections over the $20{,}000$-step replay. Over the 540 pools, 40.4% of the seeded entries are never selected again and 55% at most five times, while 20.8% are read at essentially every step. Sampled every 200 steps, 83.2% of a pool's live entries are not read again in the next 100 (§O). Yet the cold mass is not safe to drop. An entry read exactly once returns after a median of $7{,}576$ steps, and the waits are heavy-tailed at every frequency: in the class read 26 to 100 times, the 95th-percentile gap is $206\times$ longer. Even the hottest class vanishes for $16{,}942$ steps in the median worst case, longer than any look-ahead the system can carry, so no recency test separates KV that will return from KV that will not.

Insight 1: *Only the near-term working set belongs in GPU memory: a dynamic per-pool resident set, not the complete history.* The rest stays logically reachable through *offload to a complete CPU backing store*, so no destructive drop can cost the top-k selection.

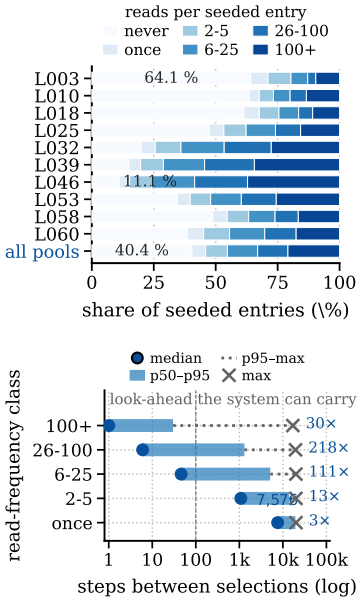


Figure 2: Dead-mass ledger. **Top**: seeded entries grouped by how often decode reads them, all pools and per layer. **Bottom**: wait between consecutive reads per frequency class: median, p50–p95, longest.

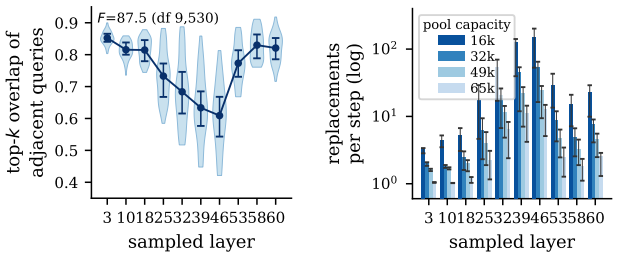


Figure 3: Demand heterogeneity across pools. **Left**: top-k overlap of adjacent queries by layer. **Right**: replacements per step by layer and capacity.

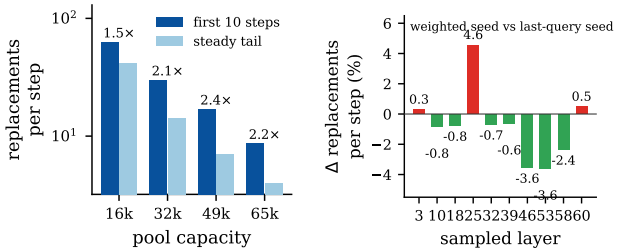


Figure 4: Admission transient and seeding. **Left**: replacements per step, first ten decode steps vs steady tail, by capacity. **Right**: change when seeding by the weighted demand vote instead of the last query's ranking. Negative is better.

Observation 2: Demand is heterogeneous and visible at admission. Demand pressure is unevenly distributed. Adjacent-query overlap of the selection falls from 0.852 at the shallowest sampled layer to 0.609 at layer 46 (Figure 3, left), and the admission-time demand signal U_j, the deduplicated union of the top-2048 of the 128 queries that precede admission, spans 4,588 to 42,373 entries (9.2×) across pools, up to 5.2× within one layer. Resizing each pool in proportion to its own U_j under a conserved total budget reduces replacements by 26.0/41.4/38.1/14.3% at mean budgets of 16/32/49/65k, close to the 27.8/44.8/43.8/24.2% of a future-knowing oracle. Where a separate KV-group axis exists, on the evaluation testbed (§5.2), depth dominates it too.

Insight 2: *Pool capacity must be dynamic, $C_j \propto U_j$, and sized at admission from the pool's own measured demand signal.*

Observation 3: Working sets drift on two clocks. On the fast clock, the selection changes quickly: step-to-step overlap averages 0.755 and falls to 0.268 1,894 steps apart. One frozen selection is already wrong for about a quarter of the next step's demand, and a pool frozen at admission still leaves a fifth of demand unheld even at the 64k design capacity (§O). On the slow clock, every capacity pays an admission transient 1.5 to 2.4× its steady level (Figure 4, left), and seeding helps little: the weighted demand vote beats last-query backfill by only 1.4% (Figure 4, right). Meanwhile, 18.7% of entries return between steps 100 and 1000, an order of magnitude past the refresh scale, so the fast mechanism must not release them (§4.1).

Insight 3: *A fast path on the scale of one to a hundred steps tracks realized demand under an LRU rule. A slow path on the scale of a thousand steps releases stale residency in bulk.*

Observation 4: The transfer path fits at pool granularity. Measured on the evaluation testbed (§5.4), the window between hidden-state availability at layer $\ell-d$ and the first read at layer ℓ is 114 to 613 μs at batch 1 (Table 11). Each DMA costs a flat 4.1 μs, so copy count, not bytes, sets fetch cost. At the design capacity each step moves only 4.3 entries per pool: one coalesced DMA per gathered pool fits the window at every measured depth, while a separate copy per token-ID run costs 1.4 to 25× it. The demand tail, a p95 at 3.8× the per-pool mean, pushes the look-ahead to two to four layers (§K).

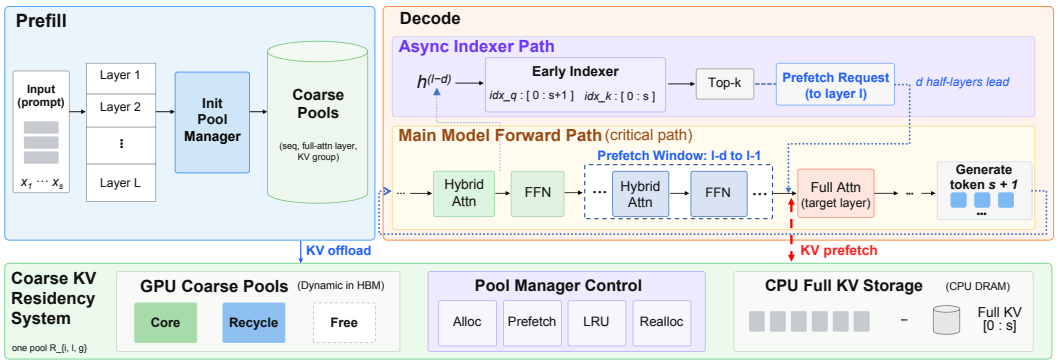


Figure 5: System placement and control order. The CPU keeps one complete KV store per (sequence, layer, KV group); the GPU keeps a coarse pool of at most $C_{i,\ell,g}$ entries per store. The read set is the top-2048 of the scored history plus the current pair, at most 2049 entries.

Insight 4: *Fetch each pool's missing entries in one gather and one coalesced DMA, scatter them on the device, two to four layers ahead of the read* (§4.2).

4 Coarse KV Residency

The system offloads every sequence's complete KV state to host memory and keeps only hot entries in coarse GPU pools for inference. The indexer's full-length key cache stays resident in HBM, and its top-k scoring runs over the full history. The design answers two questions. First, how to maintain the coarse pools so that each decode round transfers as little as possible per layer, balancing HBM footprint against PCIe traffic. Second, how to run the indexer off the critical path and open a window for CPU-to-GPU prefetch, so that each read set is in place before its attention begins.

Formally, the host stores every sequence's complete KV state $\mathcal{K}_{i,\ell,g}$, and the GPU keeps a coarse pool $R_{i,\ell,g,t} \subset \mathcal{K}_{i,\ell,g}$ of at most $C_{i,\ell,g}$ entries for each sequence, layer and KV group. Write j for the pool triple (i, ℓ, g) and drop subscripts where clear. In Figure 5, at the end of each prefill layer, warm start selects the initial coarse pool for every sequence, layer and KV group, and each decode round then maintains the pool dynamically with LRU retention. Each attention layer reads the top-k set chosen by the early indexer, leaving the top-k selection unchanged, and the pool swap completes before attention starts. Section 4.1 answers the first question: what a pool holds and how its contents change. Section 4.2 answers the second: how predicted demand arrives before its read.

4.1 Pool Residency

Admission capacity. Admission fixes each pool's capacity once, before decoding begins. Pool j's demand signal U_j counts the distinct entries selected by the last 128 prefill queries (§3.2), and the capacity is proportional to U_j under per-rank conservation:

$$U_j = \Big| \bigcup_{q \in \text{last } 128} \text{top2048}(q) \Big|, \qquad C_j \propto U_j \quad \text{subject to} \quad \textstyle\sum_{j \in r} C_j = B_r. \tag{4}$$

A sequence whose context fits $\min_{\ell,g} C_{i,\ell,g}$ stays fully resident and issues no transfer. §G gives the water-filling that realizes (4) under a per-pool floor and cap, with an integer correction that makes the rank sum exact.

Warm start. Warm start fills each pool before decoding begins. The initializer scores each entry by a weighted recent-demand vote over the prefill queries,

$$\text{score}(x) \;=\; \text{count}_{\text{last }64}\big(\,\text{top4096}\,\big)(x) \;+\; 2\,\text{count}_{\text{last }16}\big(\,\text{top2048}\,\big)(x), \tag{5}$$

and admits the highest-scoring entries first. Remaining slots are backfilled to C_j from the full ranking of the last prefill query, so every pool starts full. Ties break by ascending token id, and entries enter the initial LRU order in ascending score. By measurement (§O), the initializer omits the sink, the first positions.

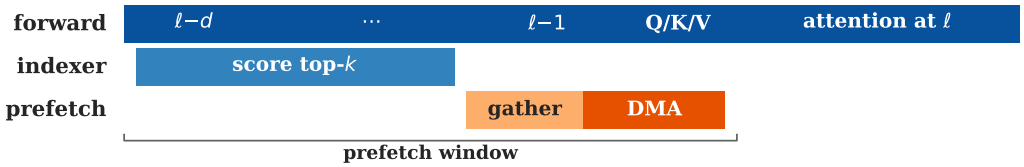


Figure 6: Schedule of one prefetch round, conceptual (no measured durations). Indexer scoring and the per-pool transfer run off the critical path and hide inside the layers that precede the target attention. The window closes after ℓ's Q/K/V, before attention starts scoring.

Decode-time LRU retention. The top-k selection reflects the model's realized demand, so LRU over selection times drives the pool's dynamic replacement. Newly generated KV stays in HBM and joins the replacement alongside the other entries in subsequent decode steps. Each round, the policy protects the selection $D_t = S_t$, inserts its missing members, and, when capacity is exceeded, evicts the unprotected resident with the oldest last-selection time:

$$\text{victim} = \arg \min_{x \in R_t \setminus D_t} \text{last_selected}(x). \tag{6}$$

Selection counts and stamps persist beyond eviction, and the reallocation below reads them at its boundaries. Algorithm 3 (§G) states the round.

Extension: periodic reallocation. Working sets drift over a long decode (§3.3). Every R=1000 steps, each pool therefore releases the entries never selected since the previous boundary, its *recycled* set, refills the freed slots in the order of (5), and keeps the top half by selection count as a protected *core* (Table 3, §L). On the replay of §5.2, this lowers replacements by 13.2% over 2,000 steps and 40.3% over 5,000 against plain LRU at the same capacity.

4.2 Early Indexer and Prefetch Path

Early indexer and lookahead prefetch. The prefetch path decouples the indexer from the main-path computation. While decoding token s, each layer's index-key cache $k'[0, s]$ over the full history already resides in HBM. For token $s{+}1$, the hidden state of layer $\ell{-}d$ is projected into the index query q_{s+1}, which scores the full $k'[0, s]$ and yields the top-k candidate set $\hat{S}_{\ell,s+1}$ over history. The tap point sits after a hybrid-attention or an FFN sublayer, so d counts residual half-blocks, and d=2 is one full block. The main-path attention then adds token $s{+}1$'s own key-value pair to the read set by construction: the read set is the top-2048 of the scored history plus that pair, at most 2049 entries (§I). A similar decoupling of the indexer selection also appears in SparDA (Fu et al., 2026). Algorithm 4 (§G) states the loop.

Prefetch window. The decoupling makes the top-k indices known before the target layer runs: after ℓ's own Q/K/V projections and before attention starts scoring, the missing entries are fetched from the CPU into the pools (Table 11). Figure 6 shows the decoupled flow.

Section 5.1 ablates the indexer lead, prices benchmark accuracy and prefetch room across d, and selects the operating point d^*; Section 5.3 reports the end-to-end decode efficiency.

System integration. All pools on a rank share one fixed GPU KV slab, with the FP8 index-key cache resident and accounted separately. One transfer round per pool is one host gather and one coalesced DMA. The design attaches to a paged serving engine (vLLM's v1 KV-connector interface) as an external connector; the engine's built-in block-hash CPU tier is whole-block by construction.

5 Evaluation

On Qwen3.5-4B we sparse-train three early-lead branches (d=1, 2, 3) with the full two-stage recipe and evaluate them end to end, taking the unmodified lead-0 branch as the baseline and adding an IndexShare branch (Z.ai, 2026; Bai et al., 2026) (reuse ratio 1:2) as a horizontal accuracy comparison (§5.1). The appendix provides further ablations on a larger model (Qwen3.6-35B-A3B) with more early-lead branches, lead 0 to 8 (§H). The evaluation covers downstream accuracy under an early indexer (§5.1), coarse-pool selection quality (§5.2), efficiency across KV offload, prefetch and end-to-end decode (§5.3), and system evidence and capacity at the operating point (§5.4).

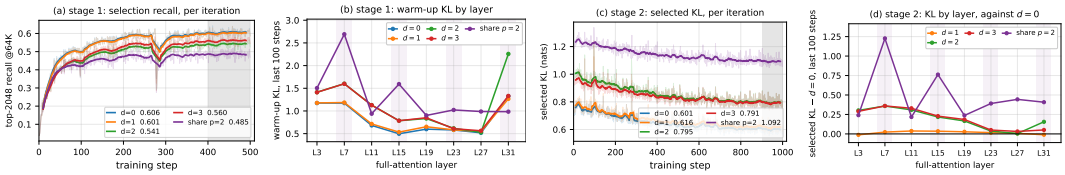


Figure 7: The five sparse-trained branches, both stages of the recipe, one curve per branch (thin: raw; thick: 20-step rolling mean; band: last 100 steps). The left two panels show S1 (indexer warm-up), the right two S2 (joint sparse training); shaded columns mark the share arm's inherited layers.

Platform. 8×NVIDIA H800-80 GB, PCIe host-to-device (measured 53.3 GB/s), vLLM 0.29.0.

Models. Five sparse-trained Qwen3.5-4B branches; the Qwen3.6-35B-A3B testbed appears only as the warm-up overlay of §5.1 and in the appendix.

Residency budget. 64K hot tokens per sequence (releasing 192K of a 256K context to CPU), matching the full-residency reference line in Figure 1a; the policy-pricing experiments of §5.2 sweep $\{16, 32, 49, 64\}$K.

Workloads. RULER at 8K to 256K (Hsieh et al., 2024), LongBench GovReport (Huang et al., 2021), and LongBench-v2 and LongCodeBench up to 256K (Guo et al., 2024; Rando et al., 2025).

5.1 Five-Branch Training, Quality, and Lookahead Fidelity

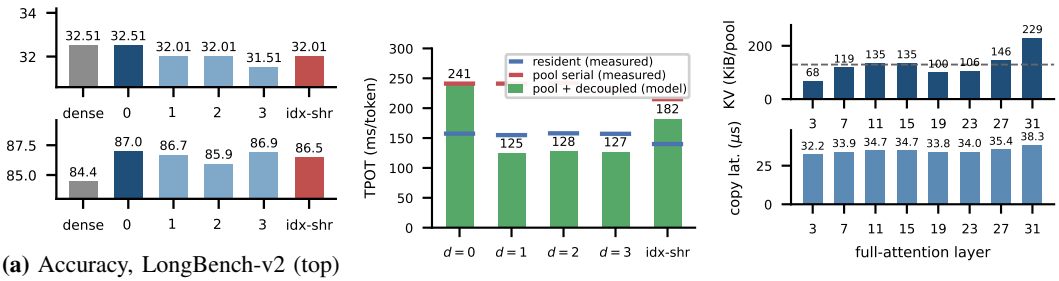
All five branches train from the same dense Qwen3.5-4B checkpoint (Figure 7): Stage 1 warms up each (layer, KV group) indexer with the trunk frozen under a full-causal KL to the teacher's scores (§H); Stage 2 unfreezes the trunk and trains end to end under top-2048-plus-current-pair sparse attention at every layer.

The branches differ only in where the index query comes from. (i) d=0 projects it from the target layer's own entry residual, the concurrent DeepSeek-style design (NVIDIA, 2026). (ii) $d\in\{1, 2, 3\}$ move it d residual half-layers early, a decoupling that also appears in SparDA (Fu et al., 2026), opening the prefetch window of Table 11. (iii) index-share runs the indexer at every second full-attention layer and hands its selection to the next (Z.ai, 2026; Bai et al., 2026); all five share the top-2048 read set.

Generated long-context quality is scored first. On the same 1700 requests (three pinned RULER NIAH tasks × 100 at 8K to 256K plus 200 GovReport rows), all five branches are decode-correct (bit-exact log-probabilities against their engine reference at 4,096 and 262,144 tokens; §5.4). The registered noninferiority gate is per-endpoint: the Bonferroni-adjusted one-sided 95% upper bound of the paired d=0-minus-candidate score drop must be at most 2.0 points on *every* one of 16 endpoints (15 RULER + GovReport), with an unresolved bound counted against the branch. Under that rule only d=0 clears all 16; d=1 clears 15 of 16, d=2/d=3 clear 12 each, and index-share clears 10 and drops GovReport 29.05→12.56. On the frozen 1,348-request decode protocol (Figure 8a, §I), the horizon arms stay within 1.0/1.05 points of d=0, and on LongCodeBench all five sit at or above the frozen dense base.

Selector-level measurements isolate the cost of acting early. Stage-1 warm-up, overlaid with the testbed's own (Figure 13, §I), shows d=1 nearly free (-0.5 to -0.8 recall points), $d\geq 2$ at 4.1 to 6.5, and index-share at -12.0 recall points warm-up, $+0.491$ nats selected KL after joint training. On unseen text the penalty is a step rather than a slope (0.79 points at d=1 against 4.74 at d=3), and held-out selected KL rises $+0.0179$ at d=1 against $+0.19$ at $d\geq 2$.

The output distribution stays close to the baseline. Every branch passes the registered output gate against d=0 (all 20 candidate-by-bucket checks on 1,616 held-out queries clear $D_{\mathrm{KL}}\leq 0.05$ nats, mean $\Delta \log p \leq 0.05$ nats, top-1 agreement ≥ 0.95), and scoring on a captured top-k set displaces the sampled action less often than redrawing the sampling noise (ρ=0.604 at temperature 1.0; Figure 14). The retained candidate set carries its own offset: scoring on it instead of all visible tokens shifts the logged action's log-probability by $+0.62$ nats on average, one-signed in 56 of 64 requests, attributed to renormalisation (§I).



(a) Accuracy, LongBench-v2 (top) and LongCodeBench (bottom).

(b) TPOT at 262K.

(c) H2D volume and copy latency.

Figure 8: Headline results on Qwen3.5-4B, H800. (a) Generated-output accuracy on the 1,348 frozen requests. (b) TPOT at 262K: decoupled-schedule model bars with the measured resident and pool-serial anchors. (c) Per-layer H2D demand over 254 decode steps at $d^*=0$. Exact values: §N.

Table 1: Held-out retention $\times$ allocation ablation, pooled over the five branches: fetched entries (percent vs uniform+LRU) on five held-out documents, four per-pool budgets. The rule set was frozen before any held-out row was read; logical flow only. Column labels: §I.

		Fetched entries, % vs uniform+LRU					
Per-pool budget	**unif.+LRU**	**unif.+LFU**	**Belady**	**U_{128}+LRU**	**U_{128}+LFU**	**fb+LRU**	**realloc+LRU**
16,384	ref	+225.45	−43.51	−2.32	+243.18	−2.20	−2.32
32,768	ref	+220.09	−47.52	−15.53	+235.63	−7.22	−15.47
65,536	ref	+133.51	−46.82	−32.56	+119.61	−14.99	−26.49
102,400	ref	+59.84	−39.55	−30.19	−15.80	−13.87	−22.29

The registered rule selects the largest horizon that clears every required level, including all 16 downstream endpoints: $d^*=0$. Selector and output fidelity alone would have admitted larger d, so the generated long-context tasks change the decision.

5.2 Coarse-Pool Algorithm Evaluation

This subsection evaluates the coarse-pool algorithm of Section 4 as logical flow under teacher-forced replay: first on the DeepSeek-V3.2-Exp capture of §3 the design was derived from, then on Qwen3.5-4B with nothing retuned.

The deployed policy sits close to the exact-read and offline-optimum references. Table 2 (Figure 15, §I) places the offline optimum, the exact read, and a recall-swept boundary forecast on the same replay: at the design capacity the forecast and the exact read move nearly the same bytes, recall loss inflates fetches only in the middle regime, and at the design period $R=1000$ the retrospective draft stays close to its period-100 flow (Table 3).

Table 1 ablates retention against allocation on held-out demand streams. U_{128} beats uniform allocation at every budget, LRU dominates LFU at small and mid budgets, periodic reallocation tracks static U_{128}, and the Belady oracle keeps a clear but bounded gap.

The dynamic allocation rule stays near the oracle across the budget sweep. Sizing pools by U_j cuts replacements by up to 41%, within 2.5% of the future-knowing oracle where the budget binds hardest (§3.2); periodic reallocation extends the gain with horizon (§4.1), and width/window sweeps keep it flat around the deployed point (Tables 6, 7, §I).

The same policy transfers to Qwen3.5-4B with nothing retuned. Replayed on 512 pools from the finished $d=0$ branch, U_{128} sizing recovers 88% of the future-knowing allocator's gain from a statistic the engine already computes at admission, and LRU remains the better ordering. Where a group axis exists it is measurable but small: the group share is 0.029, and depth dominates. The admission decisions compose: the weighted vote of (5) reduces measured K/V H2D by 26% across twenty frozen sessions with bit-identical outputs (§N). Capacities stay fixed at admission: a slot-sum-equalizing controller raises both fetches and the generation wall.

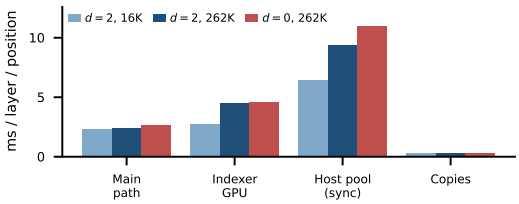


Figure 9: Decode operator groups per layer and position (H800, TP1/B1); the d=2 arm is a timing diagnostic.

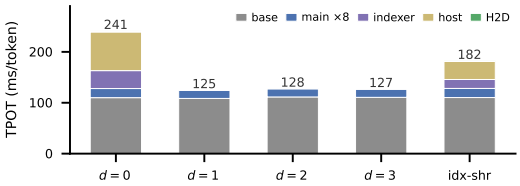


Figure 10: TPOT stage composition at 262K under the counterfactual schedule model of Figure 8b.

5.3 End-to-End Decode under KV Offload

This subsection evaluates the decode path in the deployment form: per-layer transfer demand, the operator decomposition, and the five-branch TPOT breakdown.

The deployment form's transfer demand is concentrated in a few full-attention layers. Figure 8c measures it on d=0 at 262K: a 3.4× spread across layers, the demand heterogeneity of §3.2 in bytes, with copy latency within 32–38 μs per pool and per-step volumes of a few MiB on all branches (Table 18, §K); the $d\geq 1$ branches hide this transfer inside the lead window.

The operator decomposition shows which costs grow with context. From 16K to 262K the main attention path stays flat, because the selected width remains 2048, while the indexer's top-k kernel grows about 5×, so at 262K the indexer GPU stages outweigh the four main attention kernels (Figure 9, Table 19). Only the index-Q projection can start before the target layer, so the frozen checkpoint realizes a concurrent-with schedule, its indexer stages starting at target-layer entry on a side stream (§J); the measured overlay on $d\geq 2$ hides 2.9 GB of H2D per step with tokens bit-identical to the serial control (Table 10), and d=1 likewise stays bit-identical.

Figure 8b is the headline efficiency comparison at 262K. Measured anchors bracket it: 155–158 ms/token resident against 231–242 in the deployment form (140 on index-share). Under the decoupled schedule of Figure 6, wherever a lead window exists ($d\geq 1$) the indexer side chain and the transfer fit under the pre-attention main chain and contribute zero to TPOT, landing the decoupled arms at 124.7–127.6 ms/token, 1.24× their resident anchors (Figure 10). These arms remain mechanism diagnostics, since only d^*=0 clears every downstream endpoint (§5.1); d^*=0 itself has no lead window, so its schedule model lands on the pool-serial anchor, and index-share decouples only its borrower layers (182.2 ms/token) (Table 20). At the operating point the mechanism's case is the freed residency of §5.4, not single-stream TPOT.

5.4 System Evidence and Capacity

The pool drives attention bit-exact: all ten engine-reference pairs (five branches × two context windows, 4,096 and 262,144 tokens) match on $8/8$ generated tokens' full-vocabulary log-probabilities and on $256/256$ top-2048 sets, and releasing the external GPU KV lowers the 262K run's net CUDA allocation by 6.00 GiB (Table 14, §N). Freed residency converts into concurrency. The wall it relieves is testbed characterization, measured in the appendix: the context×batch feasibility grid and the TPOT growth inside it (Table 4), the throughput value of each feasible batch doubling (Tables 5, 15), and the occupancy replay behind the 85.6% of §2 (Figure 16).

6 Conclusion

Coarse KV residency decouples logical reachability from HBM occupancy: a complete CPU backing store serves dynamic GPU pools sized at admission, and an early indexer prefetches demand off the critical path, bit-exact against the model's own sparse selection. Freed residency converts into concurrency exactly where full residency binds: a parked sequence now costs the GPU its working set, not its context.

AI USE STATEMENT

Generative AI tools were used to polish the paper's prose and to assist with code development. The authors manually reviewed all AI-assisted text and code and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies memory management for large language model inference and post-training rollouts. All experiments use publicly available model checkpoints and public benchmarks, and the production workload statistics are aggregated and contain no personally identifiable information. The study involves no human subjects. The authors have read the ICLR Code of Ethics and confirm that this submission adheres to it.

REPRODUCIBILITY STATEMENT

Section 5 fixes the evaluation setup: the hardware platform and engine version, the five sparse-trained model branches, the residency budget, and the workloads. The noninferiority gate is registered before the candidate branches are scored (Section 5.1). The appendix states the pool-maintenance and prefetch algorithms (Appendix G), and the sparse-training recipe, per-arm training detail, and evaluation instruments behind each reported number (Appendix I).

REFERENCES

Amey Agrawal et al. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve, 2024. URL https://arxiv.org/abs/2403.02310.

Yushi Bai et al. IndexCache: Accelerating sparse attention via cross-layer index reuse, 2026. URL https://arxiv.org/abs/2603.12201.

Zefan Cai et al. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling, 2024. URL https://arxiv.org/abs/2406.02069.

Xinhang Chen et al. ESS: An offload-centric latent-cache management architecture for DeepSeek-V3.2-Exp, 2025. URL https://arxiv.org/abs/2512.10576.

Yilong Chen et al. NACL: A general and effective KV cache eviction framework for LLMs at inference time, 2024. URL https://arxiv.org/abs/2408.03675.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression, 2026a. URL https://arxiv.org/abs/2609.19969.

DeepSeek-AI. DeepSeek-V4 preview, 2026b. URL https://www.deepseek.com/en/news/v4-preview/. Official announcement of DeepSeek-V4-Pro and DeepSeek-V4-Flash (284B total / 13B active parameters; DeepSeek Sparse Attention). Accessed 2026-09-19.

Jiazhan Feng et al. ReTool: Reinforcement learning for strategic tool use in LLMs, 2025. URL https://arxiv.org/abs/2504.11536.

Wei Fu et al. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning, 2025. URL https://arxiv.org/abs/2505.24298.

Yaosheng Fu et al. SparDA: Sparse decoupled attention for efficient long-context LLM inference, 2026. URL https://arxiv.org/abs/2606.04511.

Bin Gao et al. Cost-efficient large language model serving for multi-turn conversations with CachedAttention, 2024a. URL https://arxiv.org/abs/2403.19708. An earlier version was titled "AttentionStore: Cost-effective Attention Reuse across Multi-turn Conversations in LLM Serving".

Wei Gao et al. RollPacker: Mitigating long-tail rollouts for fast, synchronous RL post-training, 2025a. URL https://arxiv.org/abs/2509.21009.

Yizhao Gao et al. SeerAttention: Learning intrinsic sparse attention in your LLMs, 2024b. URL https://arxiv.org/abs/2410.13276.

Yizhao Gao et al. SeerAttention-R: Sparse attention adaptation for long reasoning, 2025b. URL https://arxiv.org/abs/2506.08889.

Yuxin Guo et al. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks, 2024. URL https://arxiv.org/abs/2412.15204.

Yefei He et al. ZipCache: Accurate and efficient KV cache quantization with salient token identification, 2024. URL https://arxiv.org/abs/2405.14256.

Cheng-Ping Hsieh et al. RULER: What's the real context size of your long-context language models?, 2024. URL https://arxiv.org/abs/2404.06654.

Cunchen Hu et al. MemServe: Context caching for disaggregated LLM serving with elastic memory pool, 2024a. URL https://arxiv.org/abs/2406.17565.

Jian Hu et al. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework, 2024b. URL https://arxiv.org/abs/2405.11143.

Luyang Huang et al. Efficient attentions for long text summarization, 2021. URL https://arxiv.org/abs/2104.02112.

Hao Kang et al. GEAR: An efficient KV cache compression recipe for near-lossless generative inference of LLM, 2024. URL https://arxiv.org/abs/2403.05527.

Kimi Team et al. Kimi Linear: An expressive, efficient attention architecture, 2025. URL https://arxiv.org/abs/2510.26692.

Woosuk Kwon et al. Efficient memory management for large language model serving with PagedAttention, 2023. URL https://arxiv.org/abs/2309.06180.

Wonbeom Lee et al. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management, 2024. URL https://arxiv.org/abs/2406.19707.

Yuhong Li et al. SnapKV: LLM knows what you are looking for before generation, 2024. URL https://arxiv.org/abs/2404.14469.

Zuning Liang et al. DualDecoder: Accelerate long context LLM inference by predictive prefetch, 2026. URL https://arxiv.org/abs/2607.26475.

Jian Lin et al. KVDrive: A holistic multi-tier KV cache management system for long-context LLM inference, 2026. URL https://arxiv.org/abs/2605.18071.

Di Liu et al. RetrievalAttention: Accelerating long-context LLM inference via vector retrieval, 2024a. URL https://arxiv.org/abs/2409.10516.

Mingjie Liu et al. ProRL: Prolonged reinforcement learning expands reasoning boundaries in large language models, 2025. URL https://arxiv.org/abs/2505.24864.

Zirui Liu et al. KIVI: A tuning-free asymmetric 2bit quantization for KV cache, 2024b. URL https://arxiv.org/abs/2402.02750.

Enzhe Lu et al. MoBA: Mixture of block attention for long-context LLMs, 2025. URL https://arxiv.org/abs/2502.13189.

Michael Luo et al. Autellix: An efficient serving engine for LLM agents as general programs, 2025a. URL https://arxiv.org/abs/2502.13965.

Xufang Luo et al. Agent lightning: Train ANY AI agents with reinforcement learning, 2025b. URL https://arxiv.org/abs/2508.03680.

Hei Yi Mak et al. HiFloat4 format for end-to-end reinforcement learning post-training of large language models, 2026. URL https://arxiv.org/abs/2607.26515.

Weian Mao et al. TriAttention: Efficient long reasoning with trigonometric KV compression, 2026. URL https://arxiv.org/abs/2604.04921.

MiniMax. MiniMax-M3. https://huggingface.co/MiniMaxAI/MiniMax-M3, 2026. Model card, June 2026. Native multimodal MoE, about 428B total / 23B active parameters, 1M-token context, powered by MiniMax Sparse Attention (MSA), a trained sparse attention operator. Accessed 2026-09-26.

MiniMax et al. MiniMax-01: Scaling foundation models with lightning attention, 2025. URL https://arxiv.org/abs/2501.08313.

Ali Modarressi et al. NoLiMa: Long-context evaluation beyond literal matching, 2025. URL https://arxiv.org/abs/2502.05167.

NVIDIA. DeepSeek Sparse Attention (DSA) — NVIDIA cuDNN frontend OSS APIs. https://docs.nvidia.com/deeplearning/cudnn/latest/fe-oss-apis/dsa.html, 2026. cuDNN documentation, accessed 2026-09-26. CuTe-DSL kernels for the DeepSeek-style sparse-attention path: dense indexer forward ($Q@K^\top$ score kernel), combined indexer forward + top-k, radix top-k, and sparse/dense backward, targeting SM90/SM100.

Matanel Oren et al. Transformers are multi-state RNNs, 2024. URL https://arxiv.org/abs/2401.06104.

Xiurui Pan et al. InstInfer: In-storage attention offloading for cost-effective long-context LLM inference, 2024. URL https://arxiv.org/abs/2409.04992.

Ruoyu Qin et al. Mooncake: A KVCache-centric disaggregated architecture for LLM serving, 2024. URL https://arxiv.org/abs/2407.00079.

Chen Qiu et al. KVDiagnosis: A diagnostic benchmark for KV-cache compression in long-context language models, 2026a. URL https://arxiv.org/abs/2608.09412.

Zhaopeng Qiu et al. FP8-RL: A practical and stable low-precision stack for LLM reinforcement learning, 2026b. URL https://arxiv.org/abs/2601.18150.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. Longcodebench: Evaluating coding llms at 1m context windows, 2025. URL https://arxiv.org/abs/2505.07897.

Guangming Sheng et al. HybridFlow: A flexible and efficient RLHF framework, 2024. URL https://arxiv.org/abs/2409.19256.

Ying Sheng et al. FlexGen: High-throughput generative inference of large language models with a single GPU, 2023. URL https://arxiv.org/abs/2303.06865.

Biao Sun et al. Llumnix: Dynamic scheduling for large language model serving, 2024. URL https://arxiv.org/abs/2406.03243.

Jiaming Tang et al. Quest: Query-aware sparsity for efficient long-context LLM inference, 2024. URL https://arxiv.org/abs/2406.10774.

Heng Wang et al. Random attention: Rethinking KV cache eviction for efficient reasoning, 2026a. URL https://arxiv.org/abs/2609.03430.

Yan Wang et al. FlashMemory-DeepSeek-V4: Lightning index ultra-long context via lookahead sparse attention, 2026b. URL https://arxiv.org/abs/2606.09079.

Can Xiao et al. OasisKV: Scaling in-decode KV cache beyond HBM with lookahead sparse prefetching, 2026. URL https://arxiv.org/abs/2608.08097.

Guangxuan Xiao et al. Efficient streaming language models with attention sinks, 2023. URL https://arxiv.org/abs/2309.17453.

Guangxuan Xiao et al. DuoAttention: Efficient long-context LLM inference with retrieval and streaming heads, 2024. URL https://arxiv.org/abs/2410.10819.

Guangxuan Xiao et al. Optimizing mixture of block attention, 2025. URL https://arxiv.org/abs/2511.11571.

Xiaomi MiMo Team. Live public stream of the MiMo-V2.6 reinforcement learning training run. https://mimo.mi.com/live, 2026. Public training dashboard, September 2026. Six-day RL post-training run with roughly 750K trajectories.

Zhiqiang Xie et al. HiSparse: Scaling sparse-attention decoding with hierarchical KV cache management, 2026. URL https://arxiv.org/abs/2608.07009.

Yi Xiong et al. LayerKV: Optimizing large language model serving with layer-wise KV cache management, 2024. URL https://arxiv.org/abs/2410.00428.

Lijie Yang et al. TidalDecode: Fast and accurate LLM decoding with position persistent sparse attention, 2024. URL https://arxiv.org/abs/2410.05076.

Jiayi Yao et al. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion, 2024. URL https://arxiv.org/abs/2405.16444.

Jiayi Yao et al. VeriCache: Turning lossy KV cache into lossless LLM inference, 2026. URL https://arxiv.org/abs/2605.17613.

Chao Yu et al. RLinf: Flexible and efficient large-scale reinforcement learning via macro-to-micro flow transformation, 2025. URL https://arxiv.org/abs/2509.15965.

Jingyang Yuan et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025. URL https://arxiv.org/abs/2502.11089.

Z.ai. GLM-5.3. https://huggingface.co/zai-org/GLM-5.3, 2026. Model card, August 2026. 753B-parameter MoE with DSA-style trained sparse attention (tag glm_moe_dsa), up to 1M-token context. No arXiv report is listed for the GLM-5.3 release. Accessed 2026-09-26.

Zhenyu Zhang et al. H_2O: Heavy-hitter oracle for efficient generative inference of large language models, 2023. URL https://arxiv.org/abs/2306.14048.

Yinmin Zhong et al. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024. URL https://arxiv.org/abs/2401.09670.

Zhengyang Zhuge et al. QUADS: Stabilizing NVFP4 reinforcement learning for MoE via QUantization-error alignment across dual sides, 2026. URL https://arxiv.org/abs/2607.15810.

Jing Zou et al. ContiguousKV: Accelerating LLM prefill with granularity-aligned KV cache management, 2026. URL https://arxiv.org/abs/2601.13631.

A Related Work

Sparse attention selects which KV entries attention reads. One family trains the sparsity into the model (Yuan et al., 2025; Lu et al., 2025; Xiao et al., 2025; MiniMax et al., 2025; Kimi Team et al., 2025; DeepSeek-AI, 2025; 2026b). A second family selects post hoc on dense models (Zhang et al., 2023; Xiao et al., 2023; Li et al., 2024; Xiao et al., 2024; Tang et al., 2024; Gao et al., 2024b; 2025b; Yang et al., 2024; Bai et al., 2026). This paper accepts the selection these methods define. Residency is orthogonal to it.

KV compression and eviction shrink what the cache holds: retention policies keep 8 to 20% of the cache at near-baseline scores (Li et al., 2024; Cai et al., 2024; Oren et al., 2024; Chen et al., 2024), quantization trades accuracy for bytes (Liu et al., 2024b; He et al., 2024; Kang et al., 2024), and lossy caches compound error as decoded length grows (Yao et al., 2026; Qiu et al., 2026a). Our candidate set stays all visible tokens. Residency drops only the local copy, never the candidate.

Offload and prefetch. Host-resident KV with on-demand delivery is established at the page level (Kwon et al., 2023), on single nodes (Sheng et al., 2023), and across clusters (Gao et al., 2024a; Qin et al., 2024; Hu et al., 2024a; Xiong et al., 2024; Yao et al., 2024; Pan et al., 2024). A recent line pairs top-k sparse attention with host-resident KV and prefetch (Xie et al., 2026; Xiao et al., 2026; Liang et al., 2026; Lin et al., 2026; Zou et al., 2026; Fu et al., 2026; Wang et al., 2026b; Chen et al., 2025; Lee et al., 2024; Liu et al., 2024a). We add the workload and the policy shape: parked RL rollouts whose KV stays materialized while they wait, served by a per-pool, dynamic residency policy.

RL rollout systems optimize execution around rollouts (Sheng et al., 2024; Hu et al., 2024b; Yu et al., 2025; Fu et al., 2025; Feng et al., 2025; Luo et al., 2025b;a; Gao et al., 2025a; Liu et al., 2025) on top of serving-side schedulers (Zhong et al., 2024; Agrawal et al., 2024; Sun et al., 2024; Qin et al., 2024). They move requests and phases. Our policy instead sets the HBM each parked sequence occupies.

B Limitations

Model scale. Both evaluated architectures are at tens of billions of parameters or below: the characterization runs on Qwen3.6-35B-A3B and the end-to-end validation on Qwen3.5-4B. The mechanism itself is not parameter-count-dependent, but the measured constants (pool budgets, lead windows, reallocation period) have not been re-measured on larger models.

Host-side cost. The CPU tier must hold every admitted sequence's complete KV state, which at the budgets of §5 is the dominant host-memory consumer, and the transfer costing prices one rank in isolation: contention for host memory and copy engines at a load-wave peak is a property of the cluster a mechanism is deployed onto, and is not modeled here.

C Offload Value of Long Requests

When a sequence's context exceeds the full-residency threshold, offloading its cold KV entries to CPU releases capacity for additional sequences. The value of offloading scales with context length: a 255K-token sequence that retains 64K tokens on GPU releases $255\text{K} - 64\text{K} = 191\text{K}$ token-equivalent capacity per full-attention layer per rank. At 1024 bytes per entry this is $\sim$186 MiB per layer per rank, or $\sim$1.8 GiB across all 10 full-attention layers on one rank, which is enough to admit approximately six 32K-token sequences at full residency.

The admission policy must therefore distinguish short and long requests. Short requests below the full-residency threshold, about 64K at the budgets we consider, are kept fully resident. They need no CPU round-trip and no indexer cost. Long requests receive a request-local budget and offload the remainder. The budget should reflect the predicted working set, not a uniform share.

The production trace shows that iteration cost is governed by the long tail, not the mean. Iteration time spans $10.1\times$ from its fastest to its slowest logged point, and the batch's *average* work does not explain it: neither the per-step mean rollout length ($r = -0.10$, n=73) nor the empty-tool-call error rate (mean 0.7% of samples, r=0.09) tracks the spread. The stage that does is rollout generation,

which spans $614\times$ and correlates with iteration time at $r{=}0.92$ over the 125-point extension (§D). This is straggler semantics: a synchronous iteration completes only when every sample finishes, so iteration time tracks the longest sample in the batch rather than the batch mean. The serving window shows the same structure at request granularity: after each load wave peaks, request volume decays over roughly the last third of the window as the longest-running sequences finish last. During the drain, concurrency is low but the resident sequences are precisely the longest ones, so per-sequence offload value is maximal exactly when spare compute and transfer bandwidth are also available. Capacity management should therefore target the p90 to p99 request, not the average request.

We stop short of quantifying that prescription. Turning it into a number needs a mixed-length admission run, one that feeds the measured per-step rollout-length distribution through an admission simulator and reports the capacity gain and total H2D cost jointly. That run requires the residency arm that does not exist yet. The per-context feasibility of Table 4 is a grid of same-length batches rather than a mixture, so extrapolating from it to a heterogeneous batch would silently assume the allocator behaves linearly in sequence length. What the capacity measurements establish is only the shape that motivates the extrapolation: the wall is crossed by the tail of the length distribution, and 71 of 73 steps have a mean, not even a p90, rollout length above the full-residency budget.

D Production Telemetry Detail

This appendix records the production-telemetry detail behind §2 and the capacity baseline of §5.4. The trace covers 73 training steps of rollout statistics, together with a window of serving telemetry from the fleet of inference engine instances that generate rollouts. The run continued training, and the telemetry now spans 125 logged points. Statistics first reported in the main text use that extension ($n{=}125$); the figures computed from the 73-step snapshot stay on that snapshot, so the two never mix inside one claim. We verified the extension is the same run deepened rather than a second workload by re-deriving every statistic the snapshot was cited for: per-step mean rollout length (95.9K), median turns per trajectory (116), trajectory totals (179K mean, 240K peak), the serving load peak ($1.8\times$ the window median) and the iteration-time spread and rank ordering all reproduce, while the iteration median rises by 16% as later, longer steps enter the sample.

Stage occupancy. The extension records what each stage of the run is doing, which the snapshot did not. The complement of the training side's busy time is the part the paper targets. Each training step reports an idle ratio that is 0% at the median, 9.5% on average, and *up to* 85.1% at its worst, while the inference engines' own idle ratio never exceeds 0.6%. The rollout side is saturated, and the training side is the one that waits. We read these as stage-level occupancy aggregates, not as a per-request concurrency measure: they say the hardware has slack, not how much of a given batch is a hole.

Length distribution and truncation. Per-step mean rollout length has a median of 97K tokens, with a range of 36 to 133K, and exceeds the 64K full-residency budget in 122 of 125 points. The within-step length distribution is what makes the budget binding: the step-wise medians of its p50, p90 and p99 are 101K, 131K and 147K tokens, and the p99/p50 ratio reaches $5.1\times$ at its widest step. Context truncation is not rare. Over the longer sample, 10 of 125 points lose more than 0.5% of their samples to the context limit and the worst reaches 12.1%, against $5/73$ and 3.9% in the snapshot.

Log-probability by length bin. Log-probability is worse for tokens in longer sequences, and it is worse as a step at the budget. The rollout token log-probability has a median of -0.134 in the 0 to 64K length bin against -0.208 in 64 to 128K, a drop of 0.075, and then plateaus at -0.199 and -0.198 for 128 to 192K and 192 to 256K. The actor model's own log-probability tracks the rollout engine's to within 0.002 at every bin. The short bin is the thinnest sample, since 23 of the 125 points carry a token below 64K, so we read the pattern as a step at the budget followed by a plateau rather than as a fitted curve.

Prefix-cache hit rate. The hit-rate panel of Figure 1 is drawn from a different sample than the rest of this appendix: per-engine telemetry over a window of about two hours at a 30-second step, 235 to 244 points for each of 88 engine-instance series. Each series' median over the whole window gives an across-engine median of 98.8% and a 10th percentile of 98.1%. Read at the end of the window instead, over each series' last 40 timestamps, two of the 88 series sit below 90% and the lowest is at 42%. Warm-up is measured the same way: every series reaches 90% and holds it for 20

consecutive samples between 6 and 13 minutes into the window, with a median of 8 minutes, which is the first tenth of it. Every quantity in this paragraph is a median over per-series medians, so it is an engine-level statistic and not a per-request hit rate.

Three boundaries qualify this extension, and all of them cut against an easy reading. Its statistical unit is a logged training step: each value is a per-step aggregate over that step's samples, so no interval below transfers to a population of requests. It is also telemetry, not a measurement of the residency mechanism. The log-probability step is an intrinsic length effect of the policy, not a consequence of KV residency. Coarse residency frees capacity; it does not make a >64K-token sequence better conditioned, and we do not count the correlation as a system benefit. The same telemetry *contradicts* the "longer and deeper is worse" slogan where it is testable. The importance sampling ratio is 0.67 for short sequences and 1.00 for every longer bin, so long rollouts are, if anything, closer to on-policy. Binning by token *position* rather than sequence length puts the worst log-probability at the beginning of the context, at -0.272 for positions 0 to 64K, rather than at depth.

Decode waste. Tokens spent on empty tool calls average 3.2% of a step's rollout tokens, with a median of 0%, but reach 71.4% on the single largest step. Re-prefilled prefix mismatches average 3.9K tokens per step, with a peak of 303K. The two percentages share the decode-token denominator, whereas prefix-mismatch tokens are re-prefill volume over a different denominator, so we quote them as separate waste counts and never as a single "wasted fraction".

No capacity counterfactual exists in the trace. The run's training batch size is *constant* on all 125 points, where min = max, while its rollout sample count ranges over 54×. Every trim/skip/carry-over counter that would record capacity pressure is zero throughout the export, covering skipped tokens, trimmed or over-length rollout chunks, and reserved carry-over samples. The run can therefore establish that the hardware is under-delivered and that the wait is on the rollout side. It cannot establish that a larger batch was blocked by HBM, because this run never tried to grow one. That causal link is therefore priced only on the testbed (Tables 4 and 5).

Rollout-bound stragglers. Iteration time spans 10.1× from its fastest to its slowest logged point at n=125, and that spread is almost entirely rollout time. The per-step rollout-generation aggregate spans 614× and correlates with iteration time at r=0.92 (Pearson; Spearman 0.73, step-level and descriptive), while the training-side consumer aggregate spans only 3.8× and correlates at r=0.19. The share of an iteration spent in rollout generation is 3.3% at the median and 54.1% at the p90, and it reaches 87.2% at its worst. The 10× jitter is therefore a handful of rollout-bound steps rather than a uniformly slower trainer. Because a fully asynchronous pipeline hides rollout inside training, a median step that is only 3.3% rollout looks cheap to optimize; the price of that hiding is the step idling of up to 85% reported above.

E What a Degraded KV State Costs

Our mechanism does not change which entries attention selects, so the evidence reviewed here bears on two things it does touch: the sparse reference we inherit, and the entries that residency leaves undelivered when a layer first reads them. The published answer on tolerance is far more permissive than context length alone suggests. Eviction retains 8 to 20% of the cache at near-baseline scores on LongBench, RULER and needle-in-a-haystack (Li et al., 2024; Cai et al., 2024; Oren et al., 2024; Chen et al., 2024). KV quantization is reported on task metrics rather than perplexity alone, with 2-bit keys and values costing on the order of two points of GSM8K (Liu et al., 2024b; He et al., 2024; Kang et al., 2024). Selective attention reaches near-full-attention accuracy while touching 1 to 3% of the stored data (Liu et al., 2024a; Gao et al., 2025b). For reasoning models this tolerance appears to strengthen rather than weaken (Mao et al., 2026; Wang et al., 2026a), and long-context benchmarks independently show that more context is not monotonically useful (Hsieh et al., 2024; Modarressi et al., 2025). RL post-training supplies the closest analogue to our own perturbation: deliberately stale asynchronous rollouts match a synchronous baseline (Fu et al., 2025), and a low-precision rollout stack that includes an FP8 KV cache preserves learning behavior (Qiu et al., 2026b).

Two limits bind us specifically. The first is that none of the above is a reward versus KV-budget curve. The tolerance evidence measures serving-time accuracy, and the RL-precision results tolerate *numeric* error rather than dropped context. The second is more dangerous: the failure mode these RL studies report is *mismatch* between the rollout and the trainer rather than loss in isolation. NVFP4

rollout against a BF16 trainer collapses after roughly 150 steps, preceded by rapidly growing rollout-trainer log-probability gaps (Zhuge et al., 2026), and raising trainer precision while leaving the rollout low-precision makes accuracy *worse* (Mak et al., 2026). Because our residency changes which entries are *available when* rather than which are *selected*, it does not inherit the compounding failure that lossy caches report as decoded length grows (Yao et al., 2026; Qiu et al., 2026a). It is also the reason the read set is never narrowed to what happens to be resident (Invariant 3 of §4): an entry skipped because it had not landed would be a load-dependent difference between the generation path and the recomputation path, precisely the divergence those studies show to be dangerous. Waiting keeps the two paths identical and leaves the exposure on timing alone, which is why §5 keeps exposed misses a separate reported quantity, and why §K prices them directly rather than subsuming them into a tolerance argument.

F Chunking Sensitivity and Offline Integrity Checks

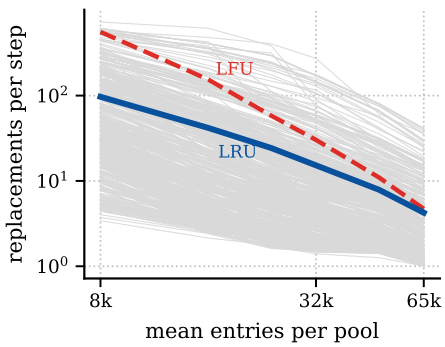
Chunking sensitivity. All three arms of the selection-vs-resampling comparison hold the window chunk size fixed, for a measured reason. On the gate cell, repeating an identical arm is bit-for-bit exact (0 of 2000 window positions differ, maximum logprob displacement 0), whereas changing only the chunk size at which the window is fed displaces the top-1 token at 3.8% to 4.7% of positions, with a mean absolute logprob shift of 0.12 to 0.16 and a 99th percentile of 1.5 to 2.1. Those displaced rows coincide only partially across chunkings, with a row-set Jaccard of 0.33 to 0.50, and they are not localized to chunk edges: the busiest eighth-of-a-chunk bin holds 21% to 22% of them, against 12.5% if they were uniform. This is therefore a reshuffling of near-tied decisions rather than a systematic offset. It is a single cell, so it prices the perturbation *class* and not its distribution.

That ladder is a property of the offline harness and must not be read as an engine noise floor: it feeds the recent window in `block`-sized chunks under an offset-causal mask, which is not what sparse decoding does. What it licenses is only the weaker statement: the harness's own numerics can displace near-tied decisions, so chunking is a controlled variable, and the ρ denominator is not a clean zero.

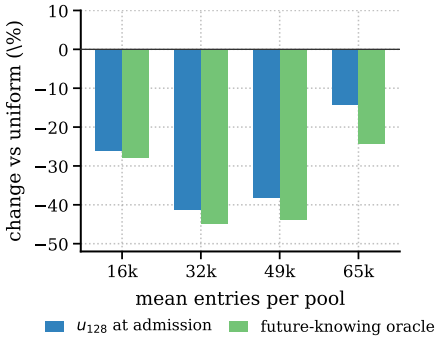
Integrity checks on the offline component. The teacher-forced replay that produces the Section 3 evidence is gated on two content assertions, checked over its full $20{,}000$-step window rather than a sample: no recorded key index may exceed the query position that recorded it, which holds with 0 violations across all 540 pools, and every row's selection must contain its own position, which it does in $539/540$ pools to within two rows of 10.8M. The second is not a redundancy check but a scope check, since a top-k that missed the diagonal would mean the recorded ranking is not the model's selection. Two further properties are enforced by construction rather than verified, and a reader of an allocation table should know which is which. Capacity is conserved up to the grid the sweep snaps to, and every arm is read at the snapped value. And there is no full-residency zero-miss control on this instrument: the recorded ranking is $64{,}000$ wide, so a pool above that width cannot be filled from it. Separately, the fidelity population's score artifacts carry their own provenance: all 320 (layer, group, context point, request) score artifacts are recomputed in FP32 with TF32 disabled and cross-checked between the CPU and GPU paths, with a maximum absolute difference of 9.54×10^{-6} and a SHA-256 recorded per artifact. That check bounds the *harness's* fidelity to the captured Q/K scores; it establishes nothing about model accuracy.

G Allocation, Warm-Start, Retention and Prefetch Algorithms

Rank-conserved water-filling. Algorithm 1 realizes (4): each pool starts at a conservative atomic-round floor, the remaining rank budget is distributed in proportion to the pool's working-set inflation U_j subject to a per-pool cap, and an integer-correction step makes the per-rank sum exact. The floor ℓ_0=6147 is the atomic-round safety bound $3\times2048+3$ (three merged queries' top-2048 plus newly generated KV); when it exceeds the average budget B, as at L=16K where B=4K, the point is flagged infeasible rather than silently under-provisioned. Runtime capacity feedback was measured on the same capture: a proportional controller adjusting every 100 steps, growing a pool by 10% and shrinking it by 2% against a target of 0.004 replacements per step. The asymmetry is the result. No pool's lowest observed capacity falls below 62.2k of the $65{,}536$-entry design point, while the wave's mean pool capacity rises to 73.2k, so the arm ratchets memory upward instead of conserving



(a) Each pool's capacity-cost curve, against uniform LFU.



(b) The U_{128} rule and an oracle, against a uniform budget.

Figure 11: Where a per-pool budget comes from: 534 scored pools, 2000 teacher-forced steps, logical counts under replay.

Algorithm 1 Rank-conserved request-local capacity allocation (U128)

Require: rank r with pools $\mathcal{P}_r$ ($|\mathcal{P}_r|$=10); per-rank budget B_r=10B, B=$\lfloor L/4 \rfloor$; last-128 prefill top-2048 selections $\{\text{pref}_q\}$; floor ℓ_0=6147; cap $u_0 = \min(L-1000, \lfloor 1.5B \rfloor)$
Ensure: per-pool budgets $\{C_j\}_{j\in\mathcal{P}_r}$ with $\sum_j C_j = B_r$

$U_j \leftarrow \left|\bigcup_{q\in\text{last }128} \text{top2048}(\text{pref}_q)\right|$ for each $j \in \mathcal{P}_r$ ▷ working-set inflation
if $\ell_0 > B$ **then return** INFEASIBLE ▷ e.g. L=16K
end if
$C_j \leftarrow \ell_0$ for all j; $\quad left \leftarrow B_r - \sum_j C_j$
while $left > 0$ **do**
  $room_j \leftarrow u_0 - C_j$; $\quad w_j \leftarrow room_j > 0\,?\,\max(U_j, \epsilon) : 0$
  $share_j \leftarrow left \cdot w_j / \sum_{j'} w_{j'}$; $\quad inc_j \leftarrow \min\big(room_j, \lfloor share_j \rfloor\big)$
  if $\sum_j inc_j = 0$ **then** ▷ integer-correction stall
    $inc_j \leftarrow 1$ for the top $left$ pools by $share_j$ among $\{room_j > 0\}$
  end if
  $C_j \leftarrow C_j + inc_j$ for all j; $\quad left \leftarrow B_r - \sum_j C_j$
end while
return $\{C_j\}$ ▷ any round whose atomic union exceeds its C_j is flagged infeasible

it and its whole wave lands at 1.12× the conserved budget. Invariant 2 excludes it on that ground, and (4) gives it no comparison at fixed B_r to be scored on.

Warm-start initialization. Algorithm 2 realizes (5): the weighted recent-demand vote fills what it can, and the admission query's own full ranking backfills every remaining slot so the pool is never left partially empty. Frequency and last-access tables are kept for non-resident entries as well and are never zeroed on eviction, which is what the reallocation policy of §4.1 reads at its boundaries.

Decode-time retention. Algorithm 3 states the per-round LRU rule of §4.1.

Gap to foresight. The rule's replacement count sits 2.5% above a future-knowing oracle's at 16k, 6.3% at 32k, 10.2% at 49k and 13.0% at the 65,536-entry design capacity, a spread of 0.42 replacements per step where a uniform pool pays 4.25 (§3.2). Simple admission-time demand inflation is therefore sufficient for the first-order capacity decision, which is why the design specifies no learned allocator; the rule's own gain falls from 41.4% at 32k to 14.3% at the design capacity, so allocation and reallocation must not both run where capacity is not scarce.

Early-indexer prefetch. Algorithm 4 states the prefetch loop of §4.2.

Algorithm 2 Initial coarse-pool selection for pool j

Require: budget C_j; prefill top-k selections pref; last prefill query's deduplicated full ranking tail; long window W_ℓ=64 (top-4096), short window W_s=16 (top-2048), weight ω=2
Ensure: resident set R_j with $|R_j| = C_j$; frequency/last-access tables F, $last$
$\mathrm{vote}(x) \leftarrow \#\{q \in \text{last } W_\ell : x \in \mathrm{top4096}(q)\} \; + \; \omega \cdot \#\{q \in \text{last } W_s : x \in \mathrm{top2048}(q)\}$
$\mathcal{V} \leftarrow \{x : \mathrm{vote}(x) > 0\}$, sorted by $\mathrm{vote}(x)$ **desc**, ties by token id **asc**
$R_j \leftarrow$ first $\min(C_j, |\mathcal{V}|)$ tokens of $\mathcal{V}$ ▷ weighted core; no fixed quota
append tokens of tail not already in R_j, in rank order, until $|R_j| = C_j$ ▷ latest-query tail fills empty slots
$F(x) \leftarrow \#\{q \in \text{last } 128 : x \in \mathrm{top2048}(q)\}$; $last(x) \leftarrow$ index of the last such q ▷ kept for non-residents; never zeroed on eviction
return R_j, F, $last$

Algorithm 3 Decode-time LRU retention (per pool j)

Require: budget C_j; initial residents R_j (Alg. 2); rounds t=1..T
Ensure: residents per round with $live \le C_j$; $last(x)$ retained for every entry seen
for each round t **do**
  $D_t \leftarrow S_t$ ▷ atomic demand; newly generated KV stays in HBM and joins later replacement
  if $|D_t| > C_j$ **then return** INFEASIBLE ATOMIC ROUND
  end if
  stamp $last(x) \leftarrow t$ for every $x \in D_t$; record a **fetch** for each $x \notin R_j$
  $remove \leftarrow \max\big(0,\ live + |\{x \in D_t : x \notin R_j\}| - C_j\big)$
  if $remove > 0$ **then**
    evict the $remove$ unprotected residents with smallest $last(x)$ ▷ LRU
  end if
  insert every $x \in D_t$ with $x \notin R_j$; $live \le C_j$
end for

H TRAINED EARLY INDEXER ON THE TESTBED MODEL

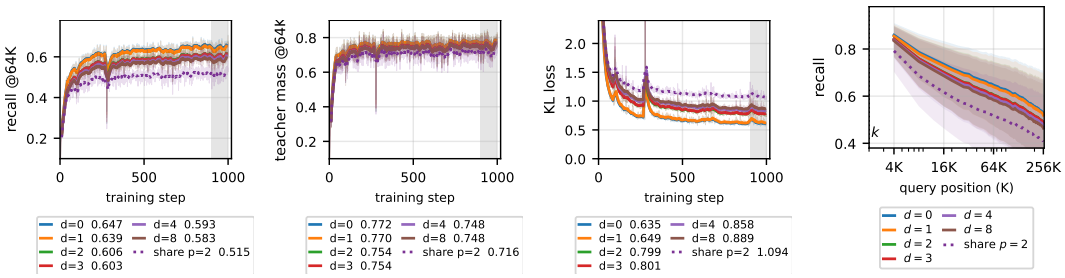
This appendix records the indexer warm-up on the testbed model, Qwen3.6-35B-A3B, which §4.2 cites as feasibility evidence. The indexer is a 64-dimensional query/key projection pair per (full-attention layer, KV group), keys held in FP8 (e4m3) with straight-through FP32 gradients, supervised by a full-causal $\mathrm{KL}(\text{teacher} \,\|\, \text{indexer})$ against the teacher's score distribution, separately for six horizons $d \in \{0, 1, 2, 3, 4, 8\}$ and one shared indexer. The shared branch keeps five indexers for the ten full-attention layers, each serving an adjacent pair and receiving equal-weight distillation from both teacher distributions it serves. Each copy saw 2.1B tokens at 262,144-token sequence length (1000 steps $\times$ 2.10M tokens). Inside every training step the indexer's top-k is compared with the teacher's own top-k at 64 query positions spaced 4096 tokens apart within the sequence, giving 80 (layer, sequence) cells per position per KV group; Figure 12 plots that series iteration by iteration, and the quoted values pool the last 100 steps (8000 cells per point, s.e.m. at most 0.0015 at 64K), because the single final-step reading sits more than 43 s.e.m. above that mean and would not be a trustworthy number.

Four quantities carry the feasibility argument. *(i) Look-ahead is cheap in accuracy*: d=1 costs 0.8 points of recall and d=3 costs 4.4, off a base of 0.647 at d=0, so within the horizons that Table 12 finds schedulable the accuracy gradient is small next to the timing constraint. The two extended horizons bound the shape beyond that: d=4 costs 5.4 and d=8 costs 6.4 points, so doubling the staleness a second time adds 1.0 point rather than another step. The shared indexer is the expensive route: it gives up 13.2 points (0.515 recall at 64K) and its KL loss lands at 1.094 against 0.635 at d=0, roughly twice the deepest horizon measured. Halving the indexer count is therefore a larger concession than any horizon this paper schedules. *(ii) Mass is concentrated where count is not*: the selected 2048 entries carry 77.2% of the teacher's attention mass at 64K and 72.5% at 128K, while agreeing with the teacher's set on 64.7% of entries at 64K and 60.0% at 128K. The disagreements therefore sit in the low-weight tail, which is what makes the recall gap of *(i)* affordable. The 128K pair is the reading §5.1 checks its renormalisation bound against. This is not a fraction of the

Algorithm 4 Early-indexer prefetch for target attention layer ℓ

Require: horizon d in residual half-layers; per-group candidate count k=2048; resident pools $R_{\ell,g}$ as of the trigger; the layer's full-length index-key cache h_ℓ (resident, §4.2); CPU backing store

1: **trigger** when the residual stream is d half-layers short of layer ℓ ▷ window opens; Table 11
2: **for** each KV group g of target layer ℓ **do**
3: $\hat{S} \leftarrow \arg\text{top-}k\big(\text{Indexer}_{\ell,g}(h^{\text{early}}_{\ell,d}, h_\ell)\big)$ over history only; $fetch \leftarrow \{\, x \in \hat{S} : x \notin R_{\ell,g} \,\}$
4: issue $fetch$ in descending predicted score as one gather + one coalesced DMA per pool, overlapped with the compute before ℓ's first read
5: **end for**
6: **at ℓ's first read, for each group g:** let $F \subseteq fetch$ be the transfers already landed
7: attend over $\hat{S} \cup \{\text{current pair}\}$; each $x \in \hat{S} \setminus (R_{\ell,g} \cup F)$ is a *late arrival*: wait on its in-flight transfer and log an exposed miss ▷ the read set is unchanged



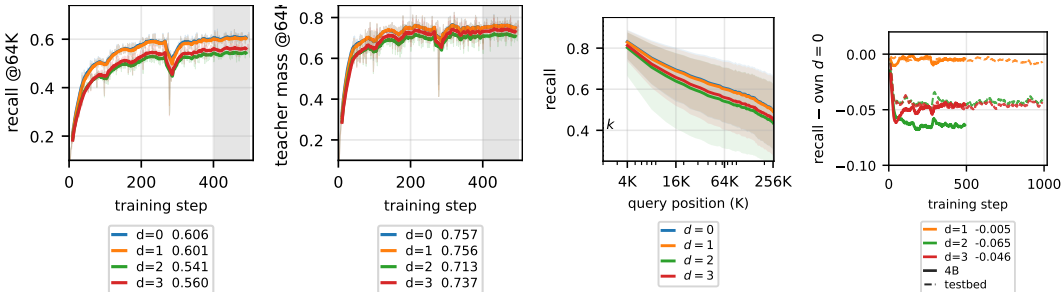
(a) Top-k recall at a 64K query position. **(b)** Teacher mass on the selected k. **(c)** Training loss, KL(teacher || indexer). **(d)** Recall vs query position, last 100 steps.

Figure 12: Trained early indexer on the testbed model, six horizons and one shared indexer; legend values pool the last 100 steps, so they are floors.

achievable: the oracle's own mass coverage at this context is not measured here. *(iii) Recall decays with available context*: 0.859 at 4K, 0.647 at 64K, 0.525 at 256K. The 4K point is near-vacuous (k=2048 of ∼4096 visible candidates), and the 256K point is this paper's binding operating point, so the honest statement is that where capacity actually binds, the trained indexer covers somewhat over half of the teacher's selection, which is a property of sparse attention at 256K, not evidence about offloading. Within the model the same heterogeneity that §3.2 measures for pools reappears in the predictor: recall at 64K ranges from 0.410 at the first full-attention layer to 0.701 at the deepest, a 1.7× spread across layers. *(iv) The warm-up has not converged*: recall rose by +1.4 to +1.8 points from the first to the last quarter of the 100-step window, across all seven branches, so these are floors for a 2.1B-token budget, not the ceiling of the recipe.

Two boundaries on this measurement. It is a train-stream diagnostic emitted inside the optimizer step with no held-out split, so it measures selection fidelity on distribution, not generalization; and it measures *selection*, not model outputs. The held-out split now exists on the second architecture (§5.1); it does not exist for this model, so this warm-up remains a train-stream figure. The training recipe is AdamW at lr=10^{-3}, weight decay 0, seed 42, 1000 steps of 2.10M tokens, one copy per branch. The d=1 copy was restarted from its step-300 checkpoint and the re-trained updates 301 to 342 reproduce that branch's own pre-kill losses digit for digit, so data addressing and RNG restoration are what fix a step's value: the seven branches are comparable iteration by iteration, not only at their endpoints.

What the warm-up recency figure does and does not license. Every flow number in the paper is computed against the demand set of a *deployed* indexer, the native-indexer capture's own top-k, so no recall multiplier is owed to them; equally, no dense-attention teacher is in that loop to say how faithful the selection is. The recall figures above therefore price the horizon choice rather than a wire volume: at r=0.647 (64K) and 0.525 (256K) an early trigger on that model would move 1.55× and 1.90× the bytes a teacher-driven schedule moves for the same useful volume. The two instruments answer different questions and their percentages must not be pooled: one prices a policy against the selection a sparse model actually makes, the other prices a predictor against a dense teacher it is being trained to imitate. A second instrument prices the same *shape* of question from the policy



(a) Top-2048 recall at a 64K query position.

(b) Teacher mass on the selected k.

(c) Recall vs query position, last 100 steps.

(d) Recall minus each model's own d=0 curve.

Figure 13: Indexer warm-up on the second architecture, one curve per horizon d; both runs consume 2.097M tokens per step.

side: Table 2 degrades a boundary forecast from recall 1.0 to 0.6 at unchanged transferred volume, on a deployed indexer's own selection, and the extra fetches land between $0.77\times$ and $1.20\times$ of the waste factor $1/r$ depending on the budget. That r is draft coverage of a demand set the indexer already defines, not agreement with a dense teacher, so the two multipliers stay unpooled. Entries the predictor omits do become transfers; they only become transfers that start too late to hide, and the selection itself is the indexer's over all visible tokens, so residency cannot narrow it.

I Evaluation Instruments and Additional Results

This appendix collects the figures and tables that §5.1 to §5.4 refer to but do not reproduce, together with the per-arm detail the main text compresses.

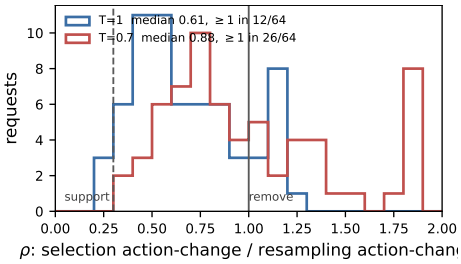
Held-out ablation column labels (Table 1). unif. = uniform per-pool budget; U_{128} = admission sizing by Eq. (4); **fb** = the conservative online feedback controller: every 250 steps it re-scores each pool by the previous epoch's low-capacity minus high-capacity logical misses and moves its capacity within a $\pm\min(16384, \text{budget}/4)$ band around the uniform budget; **realloc** = periodic full reallocation to U_{128} (§4.1); **Belady** = offline future-knowing oracle. All columns share the retention policy of their header (LRU or LFU).

Five-branch training detail. At warm-up the share arm's recall is 0.485 against 0.541 to 0.606 for the horizon arms, and its warm-up KL is 1.328 against 0.819 at d=0. After joint sparse training the share arm's margin over the worst horizon arm widens from 0.182 to 0.297 nats even though its gap to d=0 does not (0.509 at warm-up, 0.491 after). Selected KL averaged over updates 900 to 999 is 0.601/0.616/0.795/0.791 for d=0/1/2/3 and 1.092 for the share arm, against a step-to-step standard deviation of 0.025 to 0.029. The worst single layer under sharing is an inheriting one at $+1.23$ nats.

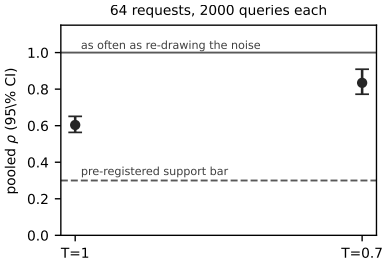
Held-out selection detail. In selected KL the horizon penalty is $+0.02$ nats at d=1 against $+0.19$ at d=3. Recall falls as context grows in all four horizon arms, from 0.849 at 8K to 0.552 at 256K for d=0. The instrument's check against the deployed kernel found 2.6×10^{-6} maximum difference in KL. On the full-window capture, the four arms score identical document sets and no scored position is a tie in any of the 44 records; the per-document standard deviations of the three deltas are 0.0026, 0.0133 and 0.0165, and the train-stream means over steps 900 to 999 are $+0.0155$, $+0.1937$ and $+0.1906$ for d=1/2/3. Layer by layer the penalty is front-loaded: at d=2 the largest layer is $+0.347$ at layer 7 and the smallest $+0.006$ at layer 27, and at d=3 layer 11 carries $+0.339$ against $+0.032$ at layer 27, while at d=1 every layer stays within $+0.038$ and two are slightly negative. The four arm means of next-token loss run 0.5997 to 0.6006.

Selection-perturbation detail. The 56-of-64 sign count of §5.1 carries a bootstrap 95% CI of $[0.48, 0.77]$. The retained mass of 0.7246 is measured on the recorded position nearest the probe's 128K centre, 131,071, taken as measured rather than interpolated.

Batch-hole detail. The 0.10 active/admitted ratio is read from the two engine views that count generating sequences directly, and the same window reports `num_requests_waiting`= 0.0, so



(a) Per-request distribution against the support and removal bars.



(b) Pooled ρ with clustered bootstrap intervals.

Figure 14: Selection-vs-resampling action-change ratio ρ over 64 requests; below 1.0 selection moves the policy less often than re-drawing noise.

Table 2: Four residency policies on one replay of the 540 measured pools. The same 2000 selection steps, the same admission seed (row 0's top-C), one boundary every 100 steps, and a budget C swept across the columns. That 100-step *boundary interval* is an instrument choice and not the round of §3.4, which is one decode step: column "f" is per pool per decode step, so it is directly comparable with that section's 4.3. "f" is positions brought across the host-device interface; "μs" prices those bytes at the measured 53.3 GB/s and adds one 4.1 μs issue per contiguous run of each fetched set, so it is what the interval's fetches cost before any coalescing across pools. Baseline (e) is offline MIN on the same stream, the fewest fetches any capacity-C residency can make: it floors fetches and not launches, which is why row (g) can land under it in μs. Every row charges every fetch, boundary or not, at the pool's own entry granularity. Two conventions travel with the table. The columns are not equivalent regimes: the demand union of one boundary interval is 8.9k positions at the median and 19.2k at the p90 over pools, so 77 of 540 intervals do not fit the 16k pool and are truncated at its boundary, which is the INFEASIBLE ATOMIC ROUND of §4.1 priced anyway to show the shape of the cliff, while 5 overflow 32k and none overflows 64k. And the (h) row's 3.8 entries per step at 64k sits 12% under the 4.3 above: same policy, same capacity, opposite tie-break among the seed, this replay evicting the tail of the admission ranking first where the earlier instrument evicted by insertion order.

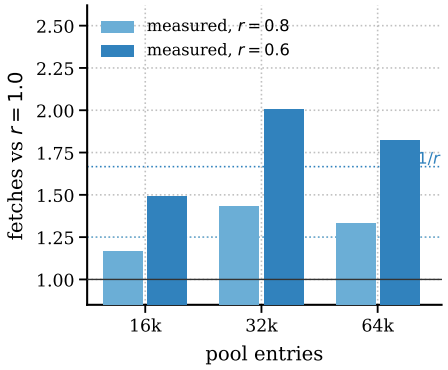
	C=16k		32k		64k	
Arm	f	μs	f	μs	f	μs
(e) offline MIN	20.9	78.7	7.9	30.2	3.0	11.7
(h) exact host read	41.4	153.1	14.3	53.8	3.8	14.7
(g) forecast r=1.0	61.6	208.4	16.7	53.7	3.9	10.2
r=0.8	72.1	246.9	24.0	82.6	5.2	18.9
r=0.6	91.9	313.1	33.6	113.8	7.1	25.5
OURS, retrospective draft	47.1	172.0	14.3	53.9	3.8	14.7

sequences idle because they are parked in tool waits rather than queued behind the engine. Block-pool utilisation averaged 7.0% over the replay.

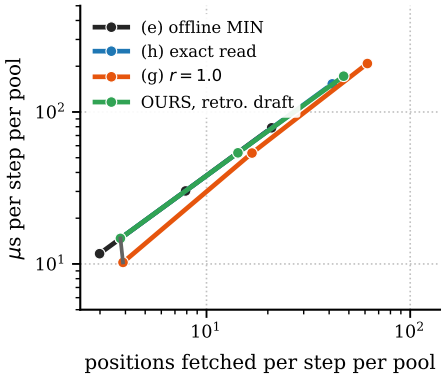
Why the current pair is read locally. The decode read set of §4.2 adds the current key-value pair to the top-2048 of the scored history. We priced that addition on all five branches over the five tuning documents, at a 262,144-token prefix and the last 128 supervised query positions, holding the prefix computation fixed and using an order-only rearrangement of the same read set as the control. The native serial selector already contains the current position at 93 to 100% of positions, because it scores at the target layer after the current key and projection exist. Forcing the current pair out of the read set raises teacher-forced NLL by 0.0038 to 0.0075 nat/token over the order control; reading it locally recovers the difference to within 0.0012, and at an equal 2048-entry budget (drop the lowest-scored history entry, add the current pair) the mean NLL stays below the history-only read set on all five branches (22 of 25 branch-document pairs), by 0.0041 to 0.0078 nat/token. The current item carries 5.4 to 5.7% of the attention mass on average. This is a tuning-document, tail-intervention

Table 3: The design's reallocation period, R=1000, on the same 540-pool replay (G4 arm). Same conventions as Table 2; the period-100 control re-derives that table's rows in the same harness. The offline floor (e) and the exact read (h) are period-invariant by construction. At R=1000 one boundary's demand union is 25.6k positions on average (20.2k median, 47.4k p90), so 366 of 540 intervals do not fit the 16k pool, 135 overflow 32k and 16 overflow 64k: the boundary forecast (g) collapses onto ≈145 fetches per step at 16k, 2.3× its period-100 cost, while the retrospective draft stays within 10% of its period-100 flow at every capacity.

Arm	C=16k f	16k μs	32k f	32k μs	64k f	64k μs
(e) offline MIN	20.9	78.7	7.9	30.2	3.0	11.7
(h) exact host read	41.4	153.1	14.2	53.8	3.8	14.7
(g) forecast r=1.0	144.4	497.5	30.0	103.3	4.5	11.8
r=0.8	145.5	502.1	31.5	111.2	5.7	19.2
r=0.6	148.8	514.5	36.3	127.8	7.3	24.8
OURS, retrospective draft	42.7	157.5	14.3	53.8	3.8	14.7



(a) Fetch cost of a recall shortfall, with the $1/r$ waste factor.



(b) Microseconds per pool per step against fetch count.

Figure 15: The four arms of Table 2 on one replay: 540 pools, 2000 steps, flow before any coalescing.

measurement; it motivates the local current-K/V read, and the generated-task consequence of the resulting decode protocol is reported separately (§5.1).

Selection-width sensitivity. The deployed candidate count is k=2048 (§4.2). Table 6 sweeps $k \in \{512, 1024, 2048, 4096\}$ on the five tuning documents of the wide top-100K archive, at admission row S=128, 2000 demand rows per pool, and conserved budgets of 16,384/32,768/65,536 entries per pool. The admission policy scales self-consistently with k: demand width, the U_j union, and the warm-start vote windows $2k/k$ all track k, while LRU retention, the water-fill floor and cap, the budgets, S, and the step count are held fixed. A hard gate validated the parameterized core first: at k=2048 on the five held-out documents with frozen caps, all 4800 cells are bit-identical to the audited k=2048 receipt before any sweep row is accepted. The measurement is flow-side only: logical fetches and replacements under replay, with no physical H2D, stall, runtime, or quality claim.

Pool-maintenance cost scales near-linearly with selection width: each doubling of k roughly doubles fetches, stable across all three budgets, so k is a direct cost dial. The U_{128} allocation benefit is present at every k but grows with the budget and is largest in the deployed region (23.3% at k=2048, 65k; 20.8% at k=4096, 65k), while the warm-start seed effect stays small ($\leq$ 3.7%). At k=4096 with the tightest budget the allocation benefit nearly vanishes (0.1%), because a pool that wide already covers the union. The sweep fixes the cost/width trade-off and shows the allocation effect is robust across k; 2048 sits where the allocation benefit is still large and the cost is half of k=4096. The quality side of the k choice is carried by the held-out selector recall of §5.1, not by this flow sweep.

Table 4: TPOT (ms/output token) for every feasible context×batch cell; "cap." marks combinations excluded by the KV block-allocation check. Qwen3.6-35B-A3B, TP=2, 2×H800, BF16, 100 forced output tokens, native MTP speculation. TPOT is recomputed from engine-internal timestamps and averaged equally over requests; `num_preemptions` is 0 in all cells here and in the 42 replication waves of Table 5. Draft acceptance spans 32 to 67% across cells, so cross-cell comparisons mix batching and speculation effects.

Context	bs1	bs4	bs8	bs16	bs32	bs64	bs128
32K	2.10	3.52	4.44	6.11	8.41	12.19	cap.
64K	2.78	4.70	4.92	6.86	9.79	cap.	cap.
128K	2.72	3.84	6.21	8.16	cap.	cap.	cap.
255K	2.84	4.99	7.03	cap.	cap.	cap.	

Table 5: Aggregate decode throughput, output tokens/s/GPU, median over three independent waves per cell (same prompts, same engine settings; wave-to-wave spread $\leq$2.1% for bs$\geq$8 and 5 to 18% at bs1, where a single request's sampling variance dominates). "cap." is the block-allocation exclusion of Table 4. The final column is the throughput gain of the *last batch doubling that is feasible*, i.e. how much a freed-up $\times 2$ of concurrency is worth at that context; the 255K entry comes from a single earlier wave (batch 4→8) rather than the replicated grid and is labelled accordingly. Workload: `SYS`.

Context	bs1	bs8	bs16	bs32	bs64	last doubling gain
32K	11.3	32.4	62.2	95.0	101.6	+7.0% (32→64)
64K	6.19	16.5	32.5	42.7	cap.	+31.7% (16→32)
128K	3.15	8.33	14.9	cap.	cap.	+78.1% (8→16)
255K	1.60	4.09	cap.	cap.	cap.	+88% (4→8, 1 wave)

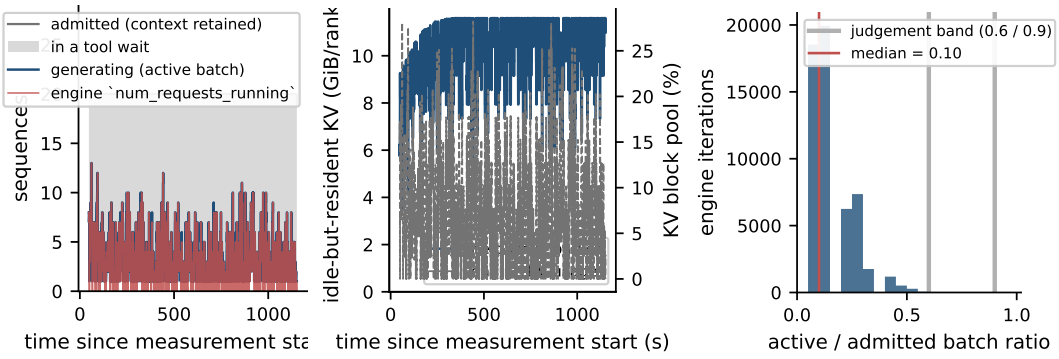
Admission union-window sensitivity. The admission signal of §4.1 unions the top-2048 of the last 128 prefill queries. Table 7 sweeps the window width $W \in \{16, 32, 64, 128\}$ on the same five tuning documents, at fixed k=2048, admission row S=128, 2000 demand rows per pool, and the same three conserved budgets. Only the admission signal changes: U_j is the union of the top-2048 over rows $S-W+1..S$, fed to the same water-filling allocation; the warm-start seed, demand width, LRU retention, floor/cap, budgets, S, and step count are held fixed. The sweep is one-sided ($W \leq 128$) because the capture's admission row S=128 bounds the available prefill history; the design point is probed from below. Two hard gates ran before any sweep row: at W=128 the regenerated caps are byte-identical to the frozen k=2048 caps, and the replay is bit-identical to the audited k=2048 receipt on all 4800 cells. Flow side only; no physical H2D, stall, runtime, or quality claim.

The allocation benefit is robust to the union window: the mean union size grows 3.4× from W=16 to W=128, yet fetches barely move, and a 16-query window already captures about 90% of the 128-query gain at the deployed 65k budget (21.1% vs 23.3%). The water-filling allocation needs the per-pool working-set spread, which saturates well before the full 128-query union, and LRU absorbs the residual; 128 is a plateau choice, not a cliff.

Five-branch downstream accuracy under the paper-consistent decode protocol. Table 8 gives the per-cell accuracies behind Table 16 (§5.1): the frozen 1,348-request campaign (403 kept LongBench-v2 rows, 100 rows above 262,144 tokens excluded, and all 945 LongCodeBench rows at up to 256K context), scored by the preregistered scorer on each branch's complete generation. Cells and request counts are the frozen plan's sample grid; the campaign's branch identifiers a0/a1/a2/a3 are the d=0/1/2/3 horizon arms and p2-a0-r2 is the index-share branch. The comparison is descriptive, with no noninferiority gate registered on this population.

J Prefetch Dependency Analysis on the Frozen Runtime

The early-indexer schedule of §4.2 moves selection work ahead of the target layer's attention. Which stages can actually move is a property of the frozen checkpoint's dataflow, not of the scheduler. Reading the frozen decode runtime (single-request path; the branch-agnostic argument is the residual



(a) Sequences generating against sequences admitted. **(b)** KV held by parked sequences against block-pool use. **(c)** Active/admitted ratio over 55,893 iterations.

Figure 16: Active-batch occupancy on the testbed (SYS: synthetic RL-shaped session replay); ratio medians 0.10 driver, 0.10 gauge, 0.15 reconstruction.

Table 6: Selection-width sweep on the five tuning documents, branch-mean total logical fetches (millions) over 2000 demand rows per pool at admission row S=128. Columns are the conserved per-pool budget. Δ_{alloc} is the fetch reduction of U_{128} allocation over uniform allocation at the same (k, budget); Δ_{seed} is the warm-start effect. Flow side only.

	Design-arm fetches (M)			**Reduction vs uniform, no seed**	
k	16k	32k	64k	Δ_{alloc}	Δ_{seed}
512	116.20	70.74	37.86	6.2/12.0/17.7%	0.7/1.7/3.7%
1024	256.12	151.34	72.29	3.3/12.6/21.0%	0.3/1.0/2.7%
2048	563.11	342.22	151.37	1.9/8.9/23.3%	0.2/0.5/1.5%
4096	1291.97	778.47	345.48	0.1/5.4/20.8%	0.1/0.2/0.8%

depth of the early source), each selector operator classifies by its earliest available input. Residual depth is counted in half-layers: depth $2i$ is block i's entry residual x, $2i{+}1$ its midpoint, $2i{+}2$ its output; the lead-d source of target layer i is depth $\max(0, 2i{-}d)$.

Two consequences for the overlap claim. First, at $d{=}1$ the window of Table 11 can host only the Q projection; at $d{\geq}2$ the wider window adds slack for the same single stage and for the transfer, because index-K and the gate still wait for x. Second, fused scoring joins the early stream with the x-derived stream, so the selection IDs cannot be issued before the target layer starts under any d. All KV in place before the target attention starts therefore requires a training-side variant that computes the gate from the early residual; the frozen checkpoint's only stream-overlap realization is the concurrent-with schedule, whose measured exposed stall is reported in Table 10 (§5.3).

K Transfer-Path Cost Ledger

This appendix carries the full cost decomposition behind §3.4: the window table, the three transfer legs at both payload shapes, the tail of the per-round demand, and the selection-kernel timing.

The raw prefetch window. Table 11 reports the window at different look-ahead depths and batch sizes: the time between hidden-state availability at layer $\ell - d$ and the attention kernel's first read at layer ℓ. These are upper bounds, taken before indexer compute, sorting, packing, and DMA startup are charged; Table 12 charges those against it.

Link rate. Pinned host→device copies reach 53.3 GB/s at the fetch shape, 2.8× what pageable buffers deliver, and keep 98% of that rate while the SMs are saturated.

Window decomposition. Table 12 charges one look-ahead window, per rank, at the two longest contexts and their largest admissible batch.

The three legs. All three legs are measured on one rank at the payload shapes this path moves, with the gathered bytes compared against the source rows at every arm. The `copies` column of

Table 7: Union-window sweep on the five tuning documents at fixed k=2048, branch-mean total logical fetches (millions) over 2000 demand rows per pool at admission row S=128. Mean U_j is the branch-pool mean union size. Δ_{alloc} is the fetch reduction of the window-W allocation over uniform allocation at the same (W, budget). The W=128 row is the deployed design point and reproduces Table 6's k=2048 row. Flow side only.

		Design-arm fetches (M)			**Allocation gain**
W	mean U_j	16k	32k	65k	Δ_{alloc} (16/32/65k)
16	11,876	562.01	344.91	155.97	2.1/8.2/21.1%
32	18,590	561.88	342.76	153.18	2.1/8.8/22.4%
64	28,048	562.09	342.08	152.25	2.1/9.0/22.8%
128	39,907	563.11	342.22	151.37	1.9/8.9/23.3%

Table 8: Per-cell generated-output accuracy (percent) of the five branches under the paper-consistent decode protocol, on the frozen 1,348-request campaign of Table 16. n is the frozen plan's request count per cell; cells are the benchmark/context buckets of the preregistered grid. Descriptive comparison.

Cell	n	d=0	d=1	d=2	d=3	index-share
`longbench_v2/16384`	22	22.73	22.73	22.73	22.73	27.27
`longbench_v2/32768`	88	42.05	42.05	40.91	40.91	40.91
`longbench_v2/65536`	72	29.17	27.78	30.56	29.17	27.78
`longbench_v2/131072`	115	25.22	23.48	26.09	25.22	27.83
`longbench_v2/262144`	106	36.79	37.74	33.96	33.96	33.02
`longcodebench/4k/4096`	5	80.00	80.00	80.00	80.00	80.00
`longcodebench/8k/8192`	7	85.71	85.71	85.71	85.71	85.71
`longcodebench/16k/8192`	2	0.00	0.00	0.00	0.00	0.00
`longcodebench/16k/16384`	83	91.57	91.57	90.36	90.36	85.54
`longcodebench/32k/16384`	3	66.67	66.67	66.67	100.00	66.67
`longcodebench/32k/32768`	181	85.08	84.53	83.98	85.08	86.74
`longcodebench/64k/32768`	33	84.85	84.85	84.85	84.85	87.88
`longcodebench/64k/65536`	133	85.71	84.96	84.21	84.96	84.96
`longcodebench/128k/65536`	1	100.00	100.00	100.00	100.00	100.00
`longcodebench/128k/131072`	275	90.18	90.55	89.82	90.18	89.82
`longcodebench/256k/131072`	31	96.77	90.32	90.32	90.32	80.65
`longcodebench/256k/262144`	191	83.25	83.25	82.20	84.29	84.82

Table 12 charges a round at the 4.1 μs issue alone, so the measured round is 2.5 to 3.9 times that charge, which makes unschedulable copies dearer still. At 28.9k entries and 37 MB, one boundary round spends 673 to 676 μs on the wire, which reproduces the measured link rate, and 110 μs scattering. Its gather leg is implementation rather than geometry: a core-parallel copy over the same payload (8 host threads) runs at 31 to 35 GB/s in 1.07 to 1.18 ms, so the gather leg costs 1.6 to 1.8 times the DMA it feeds, while a per-row indexed take of the same rows runs at 1.15 to 2.16 GB/s in 31 to 32 ms and costs 46 to 48 times. Merging the payload's contiguous runs buys almost nothing by itself: between 1.4 and 2.9 tokens per run the boundary round moves by 3 to 6%, because the copy loop rather than the span structure is what costs. At the per-step shapes the ordering reverses, since there the kernel's 3.7 μs dispatch overhead exceeds a 1.7 μs per-row take. Overlapping the gather with the link helps up to a point: the best chunk count is four at both run densities, and thirty-two chunks cost 14 to 17% *more* than no overlap at all. The residual explains where the gain stops: both legs draw on host memory bandwidth, so the copy and the transfer contend for the same resource and the round settles above $\max(\text{gather}, \text{DMA})$ rather than at it. At the 4 MiB shape overlap is worth only 8 to 10%, and at the per-step shapes nothing.

The tail of per-round demand. The tail is measurable on the testbed's own replay, where window and demand come from one model. That replay evicts as well as fills: every record carries a non-zero eviction count, a median of 1.07 evictions per fetch, so the tail below is a tail of replacement and not an artifact of a pool that never reaches capacity. Replaying retention over 64 documents at their engine's budget (20 pools per sequence, k=2048, 666 rounds of three consecutive queries per

Table 9: Dependency classification of the selector's stages on the frozen checkpoint. "Early" means startable before the target layer begins; "concurrent" means startable at target-layer entry on a side stream, alongside the main norm→QKV→norm→RoPE chain. No timing claim; this is a dataflow read of the frozen source.

Stage	Input	Earliest start
index-Q projection + norm	source residual at depth $2i-d$	early (lead half-layers before layer entry)
index-K projection + norm	target entry residual x	concurrent
W/α gate projection	target entry residual x	concurrent
index-K quantization (FP8)	projected index-K	concurrent
fused scores	early Q, x-derived K and gate	after both streams
top-k + copy/scatter	fused scores	after fused scores

Table 10: Concurrent-with overlay path on the three quality-gate-not-passed branches: engineering timing only, not a deployment speedup claim. d=1 (a1, inconclusive at the quality gate) is the rung-6 integrated overlap under the paper-consistent decode protocol, measured as event-off TPOT control vs overlay and exposed wait at consumption; d=2 (a2) and d=3 (a3) fail the downstream quality gate and enter here only as mechanism diagnostics; the operating point d^*=0 of §5.1 is unchanged. Exposed stall (wait at the target attention) and the TPOT overlay delta (overlay-path compute cost) are different readings and must not be combined; H2D traffic is decimal GB.

Branch	Context	Exposed stall p50/p95 (ms)	n	Transfer / TPOT reading
d=1 (a1)	16K	wait 0.0385/0.0411	264	TPOT control 147.1 / overlay 159.7 (ratio 1.085)
d=1 (a1)	262K	wait 0.0343/0.0449	376	TPOT control 154.2 / overlay 170.4 (ratio 1.105)
d=2 (a2)	16K pilot	0.0029/0.0061	1,024	TPOT delta +31.4 ms/token
d=2 (a2)	262K	0.0031/0.0068	4,064	2.90 GB in 357.0 ms hidden; delta +41.2
d=3 (a3)	16K pilot	0.0031/0.0062	1,024	TPOT delta +32.4 ms/token
d=3 (a3)	262K	0.0031/0.0079	4,064	2.84 GB in 361.9 ms hidden; delta +45.3

The d=1 rows are the rung-6 integrated overlap under `decode_h2048_self_v1` (`results/idxq-lead1-rung6-h2048self-20260926-015300`); generated tokens are bit-identical to the serial control at both lengths (32/32), and the exposed wait sits at the chain latency because the producer window is ≈0 on the resident runtime. The d=2 main 262K row is job `233009` (`results/idxq-lead2-prefetch-20260925-233009`), slab parity 0/4,064 mismatches; its 16K pilot is job `215430`. The d=3 main 262K row is job `215505` (`results/idxq-lead3-prefetch-20260925-215505`), slab parity 0/4,064, with 1/4,064 consumptions (0.025%) above the event resolution; its 16K pilot is job `204550`. The `kv_fully_staged_before_target_attention` property on d=2/3 is asserted only from the measured stall ≈0, not from the schedule. TPOT deltas are event-off, single-prompt readings against serial controls of 135–152 ms; they charge the overlay's own compute (slab-read attention plus gather), not the hidden transfer.

document) sums each round's fetches across the pools of the sequence; the sum reconciles to the per-pool totals, and the stored p95 equals the percentile recomputed from the saved per-round vector on all 512 records that keep it. The p95 round sits at a median of 2.7× its own mean round, spanning 1.5 to 4.5× document by document, and the median of the 48 cells (six retention policies, two score modes, four context centers from 128K to 255K) spans 2.4 to 3.0×. The worst round of a document is a median of 7.3× its mean and reaches 38.7×. Applied to the staged column of Table 12 at the cell medians, the 51 μs round at 255K and batch 8 becomes 122 to 153 μs, past the 89 μs the one-layer window leaves, and the 101 μs round at 128K and batch 16 becomes 242 to 303 μs against 158 μs. Two layers carry every cell at both contexts. The favorable end of the band, 1.5×, is the only case that still fits one layer. This is the testbed trunk, which runs without adaptation to its own sparse path, and its geometry is 20 pools rather than 61, so what transfers is the shape of the tail rather than its level. On the native-indexer capture the same per-round footing is a sharper tail at the single pool and a milder one per sequence: at the design capacity of 65,536 a pool's own step count has mean 4.26 (the 4.3 quoted in §3.4, from an independent replay), median 1, 90th percentile 7, 95th 16, 99th 64 and a worst step of 931, so the pool's 95th step is 3.8× its mean and its worst step 218×. Summed over the ten recorded layers of one sequence the same round has mean 42.6 against a 95th percentile of 154, a factor of 3.6, and that factor barely moves with the budget (3.6 to 3.9× at 16, 32

Table 11: Prefetch window (μs) at different look-ahead layers and batch sizes. Context determines the x-axis; batch size determines concurrency. Values are averages over 10 complete batches; not per-layer guarantees. It does not subtract indexer, packing or DMA overhead; Table 12 charges those against it. The "GB/s per GPU" column is the *placeholder* demand of 400 missed tokens per sequence per round divided by the 1-layer window; it is a scale marker only, and the measured demand is charged in Table 12.

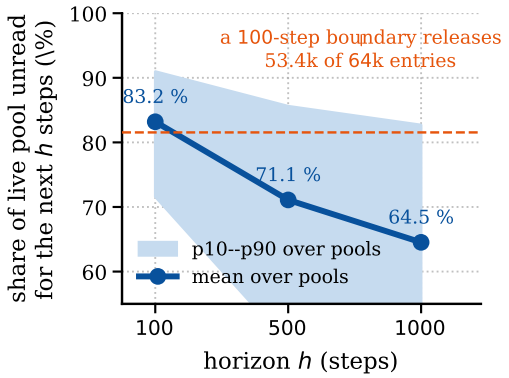
Context	Batch	1 layer	2 layers	3 layers	4 layers	GB/s per GPU
32K	1	114	228	341	470	3.6
32K	8	206	411	619	889	15.9
32K	32	366	730	1098	1729	35.8
64K	1	115	229	342	494	3.6
64K	8	224	448	673	1050	14.6
128K	1	114	228	342	534	3.6
128K	8	228	455	686	1238	14.4
255K	1	114	228	341	613	3.6
255K	8	228	456	687	1572	14.4

Table 12: Measured cost decomposition of one look-ahead window, per rank, in μs, at the two longest contexts and their largest admissible batch. "window" is the mean per-layer gap measured at each context's largest admissible batch (16 at 128K, 8 at 255K) by the same procedure as Table 11; the two tables are separate measurement waves and agree to within 4% on the one cell they share (255K batch 8: 220 μs here versus 228 μs there). "select" is the measured `torch.topk` call over 10 pools (`SYS`); it is a generic-kernel figure, and the selection-kernel paragraph below reports what the engine's two fused selection kernels cost relative to it. "residual" is window − select. "staged" is the measured demand of the 61-pool replay (262 tokens per sequence per round) moved as one contiguous run per pool at the measured 53.3 GB/s. "net" is residual − staged / residual charged with one 4.1 μs launch per owned pool instead; a negative entry means the round's fetches cannot be issued and completed before the target layer reads them. The pairing is a composite: the window is testbed engine timing and the demand is a 61-layer model at a 131K context. Both sides' tails are measured on their own footing: the replay below gives a window and a demand from one model, and the native-indexer capture gives the per-pool and per-sequence percentiles of its own demand.

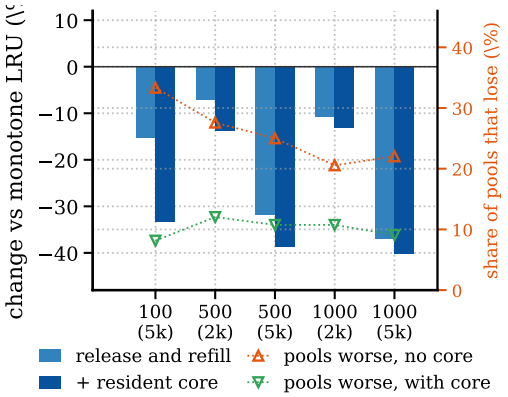
Context	Batch	Depth	window	select	residual	staged	net (staged / copies)
128K	16	1	275	117	158	101	+57 / −3844
128K	16	2	558	117	441	101	+340 / −3561
128K	16	4	1776	117	1659	101	+1558 / −2343
255K	8	1	220	131	89	51	+38 / −1912
255K	8	2	438	131	307	51	+256 / −1694
255K	8	4	1559	131	1428	51	+1377 / −573

and 65k). Cross-layer correlation of the per-round counts is 0.13, so the layers of a sequence peak nearly independently and pooling makes a round look better behaved than the pools inside it. The quantity a one-layer window must absorb is the per-pool one, 3.8×, not the testbed's 2.7×, which is a sum over 20 pools.

Selection-kernel timing. Selection over a shorter vector is the open cost item of §3.4. Timed at k=2048 over 1, 10 and 61 rows of 131,072 and 255,000 scores, under a flat and a spiked score distribution, the engine's two fused selection kernels (`top_k_per_row_prefill`, `large_context_topk`) land at 0.36× to 2.37× of the generic `torch.topk` figure the main text quotes, which is therefore a mid-range reference for a fused one. Both fused kernels also deviate from the exact selection: the per-row kernel returns 64 to 19,255 duplicate ids per call, and the large-context kernel is a bucketed approximation that keeps 96% to 99.9% of the softmax mass while returning only 12% to 31% of the exact top-2048 ids. Since invariant 1 makes the selection



(a) Unread share of the live pool over the next h steps.



(b) Change against monotone LRU per boundary period.

Figure 17: Two-zone reallocation at a fixed 65,536-entry pool, 540 pools per arm; logical counts under replay, not measured traffic.

itself the model's, not an approximation to it, the generic timing stands as the mid-range cost of an exact select, and a fused kernel that is both exact and cheaper is unmeasured.

L Periodic Reallocation: Measurement Detail

This appendix carries the measurement behind §4.1: the four offline results at full resolution, the refill's transfer cost, and the idle-share figure.

The four results. All arms run on the characterization capture at a 65,536-entry capacity, 540 pools per arm, every refill charged at pool granularity. *(i) Release must be followed by a refill.* Refilling makes the policy a win: -13.2% replacements at a 2000-step horizon and -40.3% at 5000. Releasing without refilling is the opposite trade: $+13.1\%$ replacements for 39.2 MB reclaimed per pool per layer, which is 2.4 GB per sequence over 61 layers. Release-without-refill buys admitted sequences and stands outside the traffic-saved accounting. *(ii) The core is a requirement.* Protection moves the share of pools that get *worse* than plain LRU from between 21 and 33% down to between 8 and 12%; the pooled mean improves either way, and that aggregation is the one that masks the losing pools. *(iii) The period is around* 1000 *steps, not* 100. At R=100 a boundary releases 53.4k of the 65,536-entry pool and 33% of pools come out worse, because a 100-step cycle discards precisely the 18.7-point population that returns inside that band. *(iv) The benefit is a horizon effect.* The gain triples from 2000 to 5000 steps because plain LRU degrades with run length while the re-allocated pool does not. RL rollouts are the long-horizon case, so every figure belongs to its step count. Finding (ii) establishes that a protected population must exist; it does not price its size, so the two constants of the policy are specified rather than swept.

The refill's transfer cost. Against the transfer budget of §3.4, the refill is not the cost: at R=1000 with a protected core a boundary moves 28.9k entries, 37 MB, whose measured legs come to 1.29 to 1.34 ms per pool once the gather is a copy that can use host cores and the payload is issued as four overlapped run-aligned chunks, against 1.66 to 1.75 ms for the same payload issued back to back. Either way it amortizes to 1.3 to 1.7 μs per round, under 0.8% of the one-layer window at the largest admissible batch (Table 12), and 1.9 to 2.6 times the DMA leg alone. That amortization is the replay's own accounting rather than a schedule: it charges a boundary's refill evenly across the R rounds that follow it, which holds only while the recycled slots are consumed lazily, as the measured 4.3 entries per step demands them, and not before the next read. Gathered row by row the same boundary takes 31 to 32 ms, and §3.4 prices why.

M Component Contributions

This appendix prices the mechanisms of Section 4 in one currency, as replacements added to a Uniform + LRU baseline at conserved capacity on the characterization capture. Prefetch changes

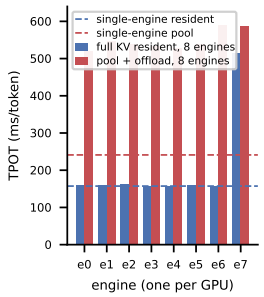


Figure 18: The coarse deployment at eight-GPU scale (Qwen3.5-4B, d^*=0, 8×H800): eight independent TP1 engines, each admitting one 262K request. Full-KV residency shows no cross-engine contention, while the pool's serial host path inflates 2.27× on one shared CPU.

only when a held entry is in hand: it moves no replacement count, so §3.4 and Section 5 price it in bytes and launches per round against the window.

Table 13: Component contributions, priced offline on the characterization capture at conserved capacity. Each row adds one mechanism to Uniform + LRU. Replacements are logical counts under replay; the horizon column names the regime where the delta lives. The last row is an excluded alternative rather than a ladder step: it holds no fixed B_r, so it has no delta in this currency.

Policy	**Replacement Δ**	**Role / regime**
Uniform + LRU	reference	baseline
+ U_{128} admission budgets	−41.4% at 32k	heterogeneous allocation; gain shrinks to −14.3% at the design capacity
+ warm start	−1.4% pooled, +0.6% steady	admission transient
+ periodic reallocation (R=1000), extension	−40.3% over 5000 steps	stale residency; long horizons
Runtime capacity feedback	excluded	the controller resizes its own budget: the wave's mean capacity totals 1.12× the conserved B_r, so its +32.1% replacements are not a comparison at fixed budget

Allocation dominates where the budget binds and buys nothing where it does not. The warm start buys 1.4% over a 2000-step replay and most of the measured traffic saving on the held-out runtime's shorter requests (§5.2). Recency beats popularity as an ordering: on the characterization capture LRU replaces 15.3 entries per step per pool against 30.5 for cumulative-count LFU at a 32k pool, a factor of 1.99, and still leads by 1.11× at the 64k design capacity (4.26 against 4.72), where both rules sit near the floor of one new key per step; the held-out architecture replicates the direction (§5.2). Runtime capacity feedback is excluded from the policy, and the exclusion is measured on both sides: the proportional controller of the characterization grid grows a pool by 10% and shrinks it by 2%, so the arm ratchets its own budget rather than conserving B_r, and the ladder cannot price it; a conserving variant, measured on the held-out runtime after U_{128} admission allocation, increases fetches at every update period and degrades every scored document (§5.2). The design ships no feedback loop. Together these rows are the argument that a measured demand statistic replaces a learned allocator: nothing is tuned per request, and every budget is one arithmetic step from the selection the engine already made. These are flow proxies at one context length on one architecture, and they set the design's parameter defaults; whether the assembled policy transfers belongs to §5.2.

N End-to-End and Admission Detail

The eight-engine coarse instance. Figure 18 measures the eight-GPU coarse form: the runtime's fail-closed guard pins each engine to TP1 with one sequence, so the eight GPUs run eight inde-

Table 14: Memory breakdown of one 262K single request admitted to the implemented engine, in GiB (1 GiB $= 2^{30}$ B; H2D flow elsewhere in this paper is decimal GB).

Component	**GiB**
Model weights	9.02
Indexer weights	0.10
Index keys + scales (FP8, full length)	0.53
Hot KV slab (2,097,152 slots)	2.00
vLLM backing cache	34.03
Other CUDA allocations	0.34
Current CUDA allocated	46.02
Reserved	47.99
Decode-phase peak	47.23
Pinned CPU prefix (host)	8.00

Table 15: TPOT (ms/output token) for every feasible context×batch cell at TP=8; "cap." marks combinations excluded by the KV block-allocation check (a conservative bound, not an observed OOM). Qwen3.6-35B-A3B, TP=8, 8×H800, BF16, vLLM 0.29.0, 100 output tokens per request (forced: `min_tokens=max_tokens=100` with `ignore_eos`), native MTP speculation (3 tokens per proposal). TPOT is recomputed per request from engine-internal first/last-token timestamps and averaged equally over requests; every feasible cell ran to completion (all 100 tokens per request), and the engine's `num_preemptions` counter is $\Delta=0$ in all nine feasible cells. Draft acceptance (accepted/drafted tokens, per-cell totals) spans 43 to 60% across cells, so cross-cell comparisons mix batching and speculation effects, as in Table 4. The allocation boundary at TP8 is 20,206 blocks per rank: 128K×64 requires 31,040, and 255K×32 / 255K×64 require 30,464 / 60,928.

Context	bs8	bs16	bs32	bs64
64K	4.24	5.23	7.09	11.50
128K	4.77	6.84	9.65	cap.
255K	5.70	8.94	cap.	cap.

pendent engines, each admitting one 262K request and decoding 128 forced output tokens. With full KV resident, seven of eight engines run at 157.5–161.7 ms/token (p50 159.1), the single-engine baseline without measurable inflation; the pool form lands every engine at 521–590 ms/token (mean 548.1), 2.27× the single-engine 241.1 ms, concentrated in the serial host path now shared by eight engines on one CPU. Aggregate throughput still grows (14.6 vs 4.1 tok/s); the decoupled schedule of §5.3 removes exactly this bottleneck. The resident bar of engine 7 includes a first request that overlapped engine start-up.

Allocation and seed compose on one request. One 196,608-token request decoded for 156 steps prices the two admission decisions together. Under uniform capacity the weighted seed lowers the mean miss from 40.7 to 22.9 per layer, group and step, and switching the capacities to U_{128}, range 31,410 to 98,304 at the conserved mean, lowers it further to 20.0, with generated tokens and log-probabilities unchanged. The run stays inside the 250-step maintenance window, so both readings price the admission decisions rather than any long-horizon maintenance effect.

Seed and allocation separate at population scale. The registered three-arm pairing runs the same teacher-forced replay on the five held-out documents of the wide top-100k capture, at admission row $S=128$ with 2,000 demand rows per pool, under one fixed-capacity LRU and three per-pool budgets, 16,384 / 32,768 / 65,536. Each pool is scored under both pairing modes so that every contrast holds the other decision fixed: swapping the last-query seed of (5) for the weighted vote, at fixed uniform capacity, lowers logical fetches by 0.14%, 0.59% and 1.90% (branch means over the five documents) and wins 70 of 75 document-budget cells; under the union128 allocation the same swap buys 0.15%, 0.71% and 3.36%. The allocation is the larger lever at every budget: union128 against uniform at fixed last-query seed saves 3.7%, 12.6% and 26.8%, and the two decisions compose near-additively to 3.8%, 13.3% and 29.3% for the paper's full admission against the uniform/last-query corner. The

seed's share grows with the budget, because a larger pool holds more of the warm-start's hot set past admission; the allocation's share grows faster, because larger pools concentrate demand where the water fill puts capacity. Logical fetches only, at the pool's entry granularity.

Transfer shape at batch. At the 65k uniform point a pool brings 98.2 entries across the interface per query row, 100.5 KB at 1,024 bytes per cached entry, which is this architecture's own geometry: head dim 256 in BF16, K and V. A batch of 32 sequences owns 1,024 pools, because 8 full-attention layers over 4 KV groups give each sequence 32 of them, so a row moves 103.0 MB, 1.93 ms of pinned host-to-device copy at the measured 53.3 GB/s. Issued as one direct-memory operation per pool the same row costs 4.20 ms of issue time at the 4.1 μs per issue §3.4 measured, and issued as one per sequence, 0.13 ms. The batch figure is that measured 512-pool mean scaled to 1,024 pools rather than a batched run, so it prices the shape of the demand and not a schedule: the transfer volume fits and the launch plan does not, which is the coalescing requirement of §3.4 re-derived on the second architecture.

Request life cycle. One transfer round per pool is one host gather, one payload H2D with its slot metadata, and a completion event; the per-round selected-slot list is a fixed $O(k)$ metadata cost paid even on a full hit. Twenty and forty serial admissions, the first one and the first two turns of each of 20 sessions, reproduce outputs exactly and pass reset and growing-length checks; across the last twenty admissions the current allocation is flat end to end (-1.1 MB) while the reserved pool grows 69 MB, the reading within this serial scope. The whole-block baseline path runs on the same engine: at 262K over 7 decode steps (224 cells) it fetches 19,563 whole blocks, 20.51 GB of block H2D, and generates identically.

Scope of the replay evidence. The Part-1 replays are teacher-forced, so the demand stream is one the model produced during prefill rather than a generated continuation, and the run is on the d=0 branch, the operating point §5.1 selects.

Exact tables behind the evaluation figures. Tables 16 and 17 give the exact accuracies of Figure 8a, Table 18 the per-layer values of Figure 8c, Table 19 the per-stage values of Figure 9, and Table 20 the per-stage values of Figures 8b and 10.

Table 16: Five-branch generated-output accuracy under the paper-consistent decode protocol (1,348 frozen requests: LongBench-v2 403 rows, LongCodeBench 945 rows, 17 context cells up to 256K). Per-cell accuracies: §I.

Benchmark	d=0	d=1	d=2	d=3	idx-share
LongBench-v2 (403)	32.51	32.01	32.01	31.51	32.01
LongCodeBench (945)	86.98	86.67	85.93	86.88	86.46

Table 17: Descriptive dense-base reference: the frozen dense checkpoint the sparse branches were trained from, run on the identical 1,348-request frozen plan and scorer with no sparse adapter. Branch rows are from Table 16.

Configuration	**LB-v2** (403)	**LCB** (945)
Dense base (reference)	32.51	84.44
d=0	32.51	86.98
d=1	32.01	86.67
d=2	32.01	85.93
d=3	31.51	86.88
index-share	32.01	86.46

O Characterization Detail for Section 3

This appendix collects the per-reading detail behind Section 3, on the same capture and conventions.

Dead-mass ledger. The ledger replays 19,999 selection steps at the 64k-entry design capacity and counts selections only. The never-read share of the seed runs from 8.1 to 66.3% at p10 to p90 across pools. It is largest in the layers that churn least: 64.1% never-read at the shallowest sampled layer, whose adjacent overlap is 0.852, against 11.1% at layer 46, whose overlap is the measured minimum at 0.609; dead share indexes how stable a layer's demand is, not how much capacity it needs. At 1285 bytes per entry on this model, the h=100 idle share is 68 MB per pool. The three live-set readings: the h=100 probe leaves about 11k of the 64k entries active; releasing the zero-selection class without refilling drains a pool to 16.7k over a 500-step window and 22.7k over a 1000-step one; the admission-time union of 128 queries' top-2048 has mean 12.2k (p10 to p90 6.1 to 20.2k).

Table 18: Physical host-to-device demand of the deployment form per full-attention layer, $d^*=0$ branch at 262K over 254 decode steps (1,016 pool queries per layer). One KV pair is 1,024 bytes (BF16 K and V, head dim 256); the volume is the measured miss count times the pair cost plus the $O(k)$ slot metadata. Copy latency is the CUDA event time of the one batched asynchronous copy per pool query; the exposed part at consumption is charged separately in Table 10. Per-step totals: 4.05 MiB KV + 0.28 MiB metadata across the 32 pools, 1.11 ms of copy time.

Layer	3	7	11	15	19	23	27	31	all
KV volume (KiB/pool)	67.7	118.8	134.8	134.9	99.7	105.7	145.6	229.2	129.5
Copy latency (μs/pool)	32.2	33.9	34.7	34.7	33.8	34.0	35.4	38.3	34.6

Table 19: Decode operator decomposition on one H800 (TP1/B1): mean ms per target full-attention layer and output position, across eight layers and 32/254 positions on one frozen natural prompt per length (16K/262K). Model stages are CUDA-event stream spans; K/V backup may include backing-path work, and host rows use wall timers. H2D/scatter rows use the paired low-marker run to avoid extra-event distortion. The $d=2$ arm is a timing diagnostic; only $d=0$ clears the registered downstream quality gate.

Stage	$d=2$, 16K	$d=2$, 262K	$d=0$, 262K
Main target-layer path			
Input norm	0.119	0.120	0.143
Q-gate/K/V projection	0.243	0.246	0.287
Q and K norms	0.233	0.235	0.283
Q and K RoPE	0.351	0.348	0.424
Current K/V backup	0.176	0.180	0.207
Four sparse-attention groups, resident	0.334	0.336	0.334
Four sparse-attention groups, pooled	0.600	0.623	0.663
Output gate and O projection	0.279	0.266	0.301
Indexer			
Index Q projection and norm	0.150	0.153	0.178
Index K projection, norm, quantization	0.123	0.126	0.146
Index W/α projection	2.022	2.025	2.036
Score against cached index K	0.010	0.067	0.067
Top-k padding	0.018	0.173	0.174
Top-k kernel	0.377	1.902	1.906
Top-k ID merge	0.036	0.040	0.040
Pool preparation and copies			
Selected-ID sync/validation (host)	1.886	3.649	3.491
LRU and slot mapping (host)	4.536	5.717	7.441
H2D copy, low-marker	0.128	0.134	0.138
KV scatter, low-marker	0.146	0.154	0.152

The 18.7 points separating $h=100$ from $h=1000$ are exactly the population that returns inside that band.

Return gaps. The heavy tail is pool-specific as well as class-wide: the widest pool in the class read 26 to 100 times exceeds 1,380$\times$ for its 95th-percentile gap over its median gap.

Position coverage. Two structural windows matter to a pool: the leading positions, which the literature keeps resident by rule, and the trailing ones, which recency keeps resident by construction. Measured against the uniform-sampling chance floor k/V_d, where V_d is the number of visible keys, a typical query's selection covers 0.919 of the last 128 positions, 54$\times$ chance, and the 128-query candidate union covers that window *completely* in all 540 pools. The same query covers 0.034 of the first 128, 2.03$\times$ chance, with 28% of the pools *below* chance there; the candidate union recovers only 0.133 of the leading window. The sink is a weak, layer-dependent and sometimes inverted effect that a demand-driven initializer mostly misses.

Overlap decay and frozen coverage. Mean overlap falls from 0.543 at 106 steps apart to 0.476 at 316 and 0.268 at 1,894; the decay is front-loaded, a mean of 0.042 points of overlap per hundred

Table 20: Five-branch decode efficiency at 262K context (one H800, TP1/B1, one frozen natural prompt, event-off TPOT ms/token). *S0* is the stock engine with the indexer serial on the critical path. *Resident*: full KV in HBM, no transfer. *Pool*: hot pool plus host offload (the deployment form), serial indexer plus exposed host path. *Model*: the theoretical decoupled schedule of Figure 6 — indexer side chain (single-row W/α and top-k, GPU slot-map/gather) and async host-to-device transfer hidden off the critical path. The model column is a counterfactual construction from audited operator spans, not an executed path (§J). d=0 has no lead window by construction, so a0 stays serial and p2's four owner layers stay serial; only p2's four borrower layers (no indexer, 14.8 ms reuse window) are fully hidden. Quality gates are unchanged: only d^*=0 clears the downstream gate; d=1 is inconclusive and $d\geq$2/index-share fail it, so their efficiency rows are mechanism diagnostics, not deployment claims.

Branch	**Schedule**	**Measured S0 TPOT**		**Model TPOT**	**H2D/step**	**H2D status**
		resident	pool	(decoupled)	(MiB)	
a0 (d=0, d^*)	serial (no window)	157.4	241.1	241.1 (unchanged)	4.54	exposed
a1 (d=1)	decoupled	154.9	241.1	**124.7**	4.51	hidden
a2 (d=2)	decoupled	157.9[†]	242.2	**127.6**	3.15	hidden
a3 (d=3)	decoupled	157.1	230.6	**126.9**	2.92	hidden
p2 (index-share)	partial	140.0	215.1	**182.2**	2.95	borrower hidden

[†]Warm-request anchor; the paired cold-start request (528.3 ms) is excluded as an initialization outlier. The a0 and p2 model rows equal their measured pool TPOTs by construction: a0's pool cost decomposes exactly into 35.2 ms serial indexer GPU + 76.0 ms exposed host path + 2.0 ms copies over the 110.0 ms non-full-attention base; p2's four owner layers carry the same serial terms (17.6 + 35.5 + 1.0 ms) while its four borrower layers contribute only the main chain. Decoupled model rows assume the parity-validated single-row W/α (CPU bit-identity) and single-row top-k (GPU token-identity); the 262K single-row top-k cost is the pre-registered rung-2 target pending direct measurement. No GPU/CPU contention is modeled.

steps over the first half of the span against 0.009 over the second, with small reversals at individual sampled lags. Frozen at admission, a pool covers a mean 0.795 of each later step's demand at the design capacity and 0.544 at an 8,192-entry pool; the worst of the 540 pools sits at 0.677.

Admission transients. Replacements per step per pool average 63.2 over the first ten steps against 41.5 in the steady tail at a 16k pool, 17.0 against 7.0 at 49k, and 8.7 against 4.0 at the design capacity. The weighted vote is the union of the previous 64 queries' top-4096 counted once and the previous 16 queries' top-2048 counted twice; it moves the pooled mean from 4.31 to 4.25 replacements per step per pool, and the two rules differ by 0.6% the other way in the steady tail.