EFFICIENT LLM INFERENCE RESCHEDULING VIA PROGRESSIVE KV CACHE MIGRATION

Anonymous authors
Paper under double-blind review

ABSTRACT

In multi-instance large language model (LLM) serving, dynamic request arrivals and unpredictable autoregressive decoding can create runtime resource inefficiencies such as GPU external memory fragmentation and load imbalance, complicating low-latency serving under high load. Runtime rescheduling can mitigate such inefficiencies by moving active requests across GPUs, but existing live-migration mechanisms require the full KV cache to be transferred before the source-side memory can be released. As a result, rescheduling decisions may take *long* to become effective, limiting their ability to promptly adjust resource allocation under evolving resource pressure. This paper revisits whether destination-side decoding must wait for full KV cache transfer. We observe temporal sparsity stability, where partial KV cache remains important over a short decoding window. With this property, the destination can resume decoding after receiving the temporally stable critical KV cache, while the remaining KV cache is refilled in the background. This enables earlier request handoff and source-side memory reclamation. So we design KVFlow, a rescheduling system for progressive KV cache migration. KVFlow exposes lightweight block-level attention statistics, selects migration-friendly requests with stable critical KV cache, and progressively refills the remaining KV cache after handoff. Our evaluation shows that, compared to a state-of-the-art rescheduling system, KVFlow reduces migration handoff latency by up to 6.89× and lowers migration-time KV cache footprint by about 30%, while preserving generation quality. In the main mixed-workload evaluation, compared with Llumnix, KVFlow reduces mean time-to-first-token (TTFT) by up to 24.0% and reduces P99 request latency and P99 TTFT by 27.1% and 47.3% on average, respectively.

1 INTRODUCTION

In multi-instance large language model (LLM) serving, dynamic request arrivals and unpredictable output lengths cause resource demand to evolve throughout execution, making it difficult to maintain low serving latency and efficiently utilize GPU memory under high load (Qin et al., 2025; DeepSeek-AI, 2024; Sun et al., 2024; Tan et al., 2025). As requests finish at different times, free GPU memory can become unevenly distributed across instances: no single instance may have enough memory to admit a new request, even when aggregate free memory is sufficient. Meanwhile, requests that generate longer outputs than anticipated continue to accumulate KV cache, potentially pushing instances toward their memory limits even when their initial memory utilization was low. Consequently, dispatch decisions based on resource availability at request arrival may become suboptimal as execution progresses, motivating runtime rescheduling to redistribute requests across instances.

Rescheduling a running request across instances requires migrating its execution state, which is dominated by the KV cache. To minimize service interruption, Llumnix (Sun et al., 2024) adopts live migration: the request continues decoding on the source instance while its KV cache is incrementally transferred to the destination. However, near-zero downtime does not imply fast migration completion. The destination can take over decoding, and the source can release the request's KV cache, only after the full cache, including newly generated entries, has been transferred. Our experiments (§3.1) show that migrating a 32K-context request on Llama-3.3-70B (Meta Llama, 2024) takes about 16.7 seconds. This prolonged completion time delays the benefits of defragmentation, load balancing, and out-of-memory (OOM) prevention: queued requests remain blocked while the

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.

source retains both the migrated request's decoding load and KV cache. Meanwhile, transferred KV cache temporarily occupies memory on both instances, consuming capacity needed by other requests in memory-bound serving. Slow migration therefore postpones resource relief while prolonging additional memory consumption.

To address these limitations, we revisit the key assumption behind live migration: *is full KV cache indeed needed?* Our analysis suggests that near-term decoding can proceed with a critical subset. However, KV importance can change as decoding proceeds (Lee et al., 2024; Sun et al., 2025), so the subset must remain useful until refill. We call this persistence *temporal sparsity stability*. While spatial sparsity identifies important KV entries at a given decoding step (Jiang et al., 2024; Tang et al., 2024), temporal sparsity stability concerns whether the selected entries remain useful across the migration window. A subset suitable for one step may therefore be insufficient for early handoff across multiple steps. This creates an opportunity for progressive migration: hand off decoding with critical KV cache, then refill the remaining cache in the background.

Realizing this opportunity requires coordinating partial-cache decoding with asynchronous refill, selecting requests that provide timely memory relief while maintaining temporal stability, and obtaining the attention statistics needed for these decisions with little decoding overhead. We design KVFlow, an LLM rescheduling system based on progressive KV cache migration (§4). Its migration protocol coordinates early handoff, background refill, and incremental source-side memory release. Temporal Importance inDEX (TiDEX) selects requests and critical subsets using migration benefit and temporal stability. A lightweight FlashAttention extension supplies block-level attention information through compact statistics collection and on-demand reconstruction.

We implement KVFlow on top of Llumnix, using a modified vLLM runtime as the execution backend. We evaluate KVFlow across multiple model scales and workloads on a 24-GPU RTX3090 cluster and an 8-GPU A800 server. Compared with Llumnix, KVFlow reduces handoff latency by up to 6.89×, lowers migration-time KV cache footprint by about 30%, and releases source-side KV cache 1.96–4.49× faster. In the main evaluation, these benefits translate into improved end-to-end serving performance under high load: mean time-to-first-token (TTFT) decreases by up to 32.9% over vLLM and 24.0% over Llumnix, while P99 TTFT decreases by 47.3% on average compared with Llumnix. Across the evaluated settings, KVFlow maintains near-lossless generation quality with negligible time-per-output-token (TPOT) overhead.

2 Background

2.1 Multi-Instance LLM Serving

Modern LLM inference services are typically deployed as multiple serving instances across GPU clusters (Yu et al., 2025; Jiang et al., 2025; Miao et al., 2024; Zhang et al., 2025). Each instance hosts the model and serves requests, while a cluster-level scheduler dispatches incoming requests according to instance load and available resources (Zheng et al., 2024; Kwon et al., 2023). After a request is dispatched, prefill processes its input tokens and stores their keys and values as KV cache. Autoregressive decoding reuses these cached states to avoid recomputing keys and values for earlier tokens, while appending new key-value pairs. The request's KV cache therefore grows as generation proceeds, increasing its memory requirements after the initial dispatch decision.

2.2 KV Cache Live Migration

As request memory footprints continue to evolve after admission, initial dispatch decisions can become suboptimal. Memory pressure can become uneven across instances, causing some instances to approach OOM while others remain underutilized. Free memory may also be spread across instances such that no single instance can accommodate a new request, even when the cluster has sufficient aggregate free memory. Existing systems mitigate OOM risks through preemption and recomputation (e.g., vLLM (Kwon et al., 2023)) or by offloading KV cache to CPU or SSD (Sheng et al., 2023; Gao et al., 2025; Xu et al., 2024). While these mechanisms relieve memory pressure within individual instances, they do not directly address the imbalance across instances. This gap motivates runtime request rescheduling, which redistributes running requests across instances to relieve memory pressure, balance load, and facilitate the admission of new requests.

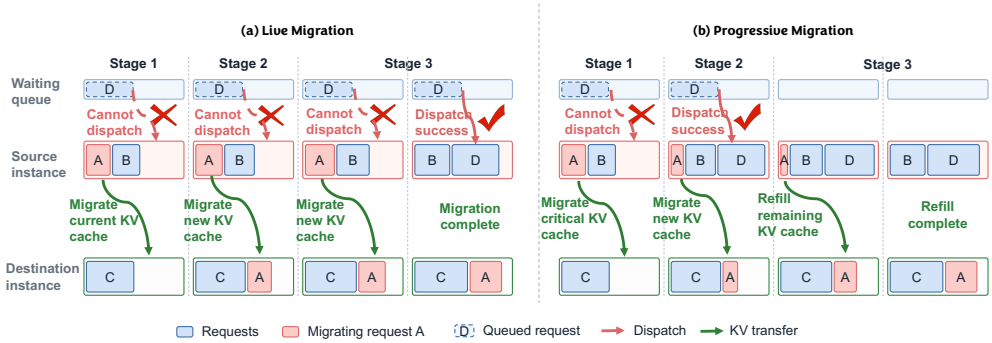


Figure 1: Illustration of live migration (left) and progressive migration (right) to solve memory fragmentation. The migration begins at Stage 1 and completes at Stage 3.

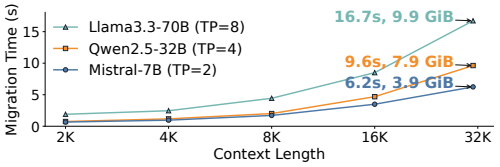


Figure 2: Completion time of existing KV cache migration across different context lengths.

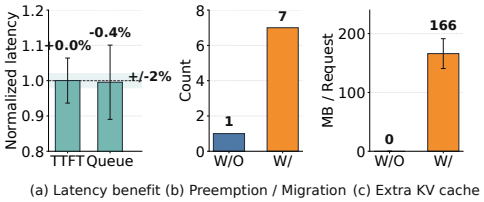


Figure 3: Live migration brings negligible latency improvement but incurs extra overhead.

Migrating a running request transfers its execution state from a source instance to a destination instance, with the KV cache as the dominant state during decoding. Stopping the request for the entire transfer would interrupt token generation until the destination resumes execution. This interruption is called *downtime*. To reduce downtime, Llumnix (Sun et al., 2024) adopts a live migration approach inspired by virtual machine migration (Clark et al., 2005), copying state while execution continues. As illustrated in Figure 1, the source continues decoding while transferring its existing KV cache to the destination. Newly generated KV cache is tracked and synchronized in subsequent rounds. At handoff, the source briefly pauses decoding to synchronize the final incremental state; once the full cache is available, the destination takes over and the source releases the request's cache. Although overlapping transfer with decoding keeps downtime short, reclamation of KV-cache memory at the source must still wait until the entire transfer is complete. §3.1 examines the resulting delay and memory overhead.

3 Motivation

3.1 Live-migration Takes Long to Complete

Figure 2 reports Llumnix live-migration completion times across models and context lengths; Appendix D.1 provides the experimental setup. At a context length of 32K tokens, migration takes 6.2 s for Mistral-7B, 9.6 s for Qwen2.5-32B, and 16.7 s for Llama-3.3-70B. Unlike a static buffer transfer, live migration must also synchronize new KV-cache entries generated as the source continues decoding. Page-based transfer can introduce additional overhead from metadata management, synchronization, and resource contention with decoding. Although live migration allows decoding to continue during KV-cache transfer, the source cannot release the request's KV cache until migration completes. This delay limits the effectiveness of runtime rescheduling in two ways.

Delayed Rescheduling Benefits. Until handoff, the request's KV cache remains at the source, which also continues decoding the request. Migration therefore provides no immediate relief from memory or decoding load at the source. Requests waiting for source-side memory can therefore remain blocked even after migration begins (Figure 1). During the high-load interval shown in Figure 3(a), average queueing latency and TTFT remain similar with and without migration. These

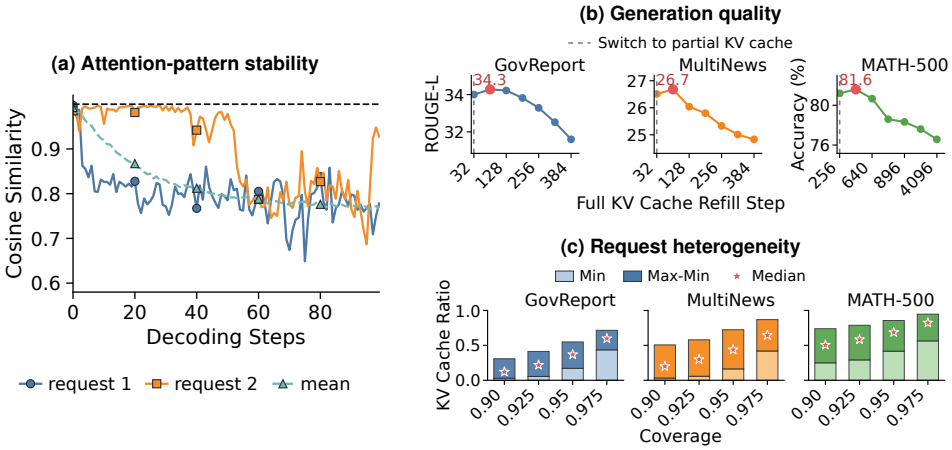


Figure 4: Opportunities for progressive KV cache migration. (a) Attention-pattern similarity between partial- and full-KV decoding on Multi-News with Mistral-7B. (b) Generation quality versus the full KV cache refill step; gray dashed lines mark the switch to partial KV decoding. (c) KV cache ratios required to reach each attention-weight coverage target across requests.

results indicate that, in this setting, rescheduling provides limited immediate benefit because source-side resources remain occupied until handoff.

Additional Memory Consumption. During live migration, KV-cache entries copied to the destination remain at the source until handoff, temporarily occupying GPU memory on both instances. In the workload shown in Figure 3(b), the system triggers seven live migrations to avoid a single preemption; Figure 3(c) reports an average of 166 MB of additional GPU memory usage per migration. Longer migration times extend the period of duplicate memory occupancy, adding pressure to an already memory-constrained system. Both delayed memory relief and temporary duplication arise because handoff and source-side cache release must wait for the full KV cache to be transferred.

3.2 Opportunity: Progressive Migration

We examine whether temporary decoding with partial KV cache preserves generation quality and how critical-cache requirements vary across requests. We emulate progressive migration by decoding with selected KV pages and restoring the omitted cache at a specified decoding step. We evaluate Mistral-7B on the GovReport and Multi-News summarization tasks using ROUGE-L, and Qwen3-4B-Thinking on the MATH-500 mathematical reasoning benchmark using accuracy. Unless otherwise specified, we retain the top 25% of KV pages ranked by page-level attention weights and compare partial-cache decoding with full-cache inference.

Decoding with Partial KV Cache Exhibits Temporal Sparsity Stability. We examine temporal sparsity stability by measuring cosine similarity between partial-cache and full-cache attention distributions across decoding steps. As shown in Figure 4(a), mean similarity remains above 85% for the first 20 iterations, then gradually decreases, although some requests retain high similarity for tens of steps. These results suggest that selected critical KV pages remain relevant over successive decoding steps, allowing partial-cache decoding to approximate full-cache attention for a limited period. We further examine whether this attention-pattern stability preserves generation quality by varying when the omitted KV cache is restored (Figure 4(b)). The results show that delaying refill can degrade generation quality. At the latest refill point, generation quality drops by 7.05%, 9.31%, and 9.43% on the three workloads, respectively. In contrast, generation quality remains close to the full-cache baseline when the remaining KV cache is restored early. This suggests that partial KV cache should serve as a temporary bridge rather than a replacement for full KV cache.

Progressive Migration Benefits are Request-Dependent. We next examine how critical-cache requirements vary across requests. Instead of fixing the retained KV-cache ratio, we measure the minimum ratio required for each request to reach a target attention-weight coverage at the migration point. We consider four coverage targets and report the highest and lowest required ratios across

requests in each dataset. As shown in Figure 4(c), these ratios vary substantially across requests, suggesting that the retained-cache ratio should be adapted to individual requests. For a given total cache size, a lower required ratio reduces the amount of data that must be transferred before handoff. Request selection should therefore consider both the size of the critical subset and the amount of source-side memory that migration can release to meet the current rescheduling need.

4 KVFLOW DESIGN

KVFlow connects migration decisions to execution through three components. The attention extension collects compact statistics during normal decoding and reconstructs block-level weights when migration is triggered. TiDEX uses these weights to select a request and its critical subset based on migration benefit and temporal stability. The migration protocol then executes early handoff and background refill, releasing source-side blocks as the destination incorporates them. Appendix A.1 presents the system architecture.

4.1 PROGRESSIVE KV CACHE MIGRATION

The migration protocol ensures that each request executes on only one instance at any given time and maintains consistent KV block mappings throughout migration. It proceeds through three stages: critical KV transfer, request handoff, and remaining KV refill, as illustrated in Figure 1(b).

Stage 1: Critical KV Cache Migration. The source transfers the critical KV subset selected by TiDEX while continuing inference. During this stage, only the source executes the request; the destination prepares the received blocks for handoff.

Stage 2: Request Handoff. The source briefly pauses the migrating request and transfers the incremental KV cache generated during Stage 1, together with the necessary request metadata. Once the destination has installed the critical subset, incremental KV cache, and metadata, it takes over decoding and the source relinquishes execution of the request. The source can then release its local copies of the KV-cache blocks incorporated at the destination, providing memory relief before migration completes.

Stage 3: Remaining KV Cache Refill. The destination continues decoding while the remaining KV-cache blocks are transferred in the priority order produced by TiDEX. Because this order may differ from token sequence order, arriving blocks must be mapped to their original logical positions even as decoding advances. The destination first receives each batch into temporary KV storage. At an engine-step boundary, it updates the request's block table to map each received block to its original logical position, using PagedAttention's separation of logical indices from physical storage. Once the batch is incorporated, the source releases the corresponding blocks. Appendix A.2 provides the protocol diagram and further block-table details.

4.2 MIGRATION SCHEDULING WITH TIDEX

KVFlow makes migration decisions at two levels: a global controller identifies source–destination pairs from instance load and KV cache capacity, and the local scheduler uses TiDEX to select requests and critical subsets. Beyond request length or queue status (Sun et al., 2024), TiDEX considers the transfer-cost variation observed in Figure 4(c) and temporal stability.

TiDEX estimates migration benefit by comparing useful memory relief against pre-handoff transfer cost. Let N_r be request r's KV block count and R_s the source's release target. We cap useful relief at $\min(N_r, R_s)$; this potential relief is realized progressively as source blocks are released, rather than entirely at handoff. To estimate transfer cost, TiDEX ranks blocks by current attention weight and selects the smallest subset reaching cumulative coverage ρ. The remaining blocks retain the same attention-based ordering for background refill. Denoting the critical-subset size by $K_\rho(r)$, TiDEX scores each request as:

$$\text{Benefit}(r) = \frac{\min(N_r, R_s)}{K_\rho(r)}. \tag{1}$$

Among high-benefit candidates, TiDEX assesses temporal stability by comparing reconstructed block-level attention weights at the current step, A_T, and N decoding steps earlier, A_{T-N}. The

top-K_ρ hit rate measures overlap between the two top-K_ρ block sets, while JS-shift is the Jensen–Shannon divergence between the two normalized block-level attention distributions. Set overlap captures membership persistence, whereas JS-shift detects redistribution of attention mass even when the selected blocks overlap. Using both metrics avoids relying on membership or weight distribution alone. A request must satisfy both the hit-rate and JS-shift thresholds to pass the stability check. Appendix B provides the benefit-analysis and stability-computation examples.

4.3 Lightweight Block-level Attention Extension

TiDEX requires attention weights to select critical KV cache and assess temporal stability. However, optimized kernels such as FlashAttention (Dao et al., 2022; Dao, 2023) and FlashInfer (Ye et al., 2025) avoid materializing attention matrices; exporting token-level weights would add memory traffic and storage overhead. To obtain these statistics efficiently, KVFlow records compact block-level ratios from FlashAttention's online-softmax intermediates and reconstructs globally normalized weights only when migration is triggered. Algorithm 1 presents both procedures for a single query and attention head.

Algorithm 1 Block-level statistics collection and on-demand reconstruction

Require: A single query/head: $Q \in \mathbb{R}^{1\times d}$, $K, V \in \mathbb{R}^{N\times d}$; KV block size B_c.
1: **During statistics collection (inside the attention kernel):**
2: Partition K, V into $T_c = \lceil N/B_c \rceil$ blocks; load Q into SRAM.
3: Initialize $O_0 = 0$, $l_0 = 0$, $m_0 = -\infty$, and $W \in \mathbb{R}^{T_c}$.
4: **for** $i = 1, \ldots, T_c$ **do**
5: Load K_i, V_i into SRAM; compute $S_i = QK_i^\top$.
6: $m_i = \max(m_{i-1}, \max(S_i))$, $\tilde{P}_i = \exp(S_i - m_i)$.
7: $P_i = \sum \tilde{P}_i$, $l_i = \exp(m_{i-1} - m_i) l_{i-1} + P_i$.
8: Record $W_i = P_i/l_i$.
9: $O_i = \exp(m_{i-1} - m_i) O_{i-1} + \tilde{P}_i V_i$.
10: **end for**
11: Write $O = O_{T_c}/l_{T_c}$ and W to HBM.
12: **When migration is triggered (outside the attention kernel):**
13: **for** each recorded ratio snapshot W needed by TiDEX **do**
14: Set $T_c = |W|$; $R_{T_c+1} = 1$
15: **for** $i = T_c$ downto 1 **do**
16: $A_i = W_i R_{i+1}$
17: $R_i = R_{i+1}(1 - W_i)$
18: **end for**
19: Supply reconstructed weights A to TiDEX.
20: **end for**

In-kernel Ratio Collection. Let the KV cache contain T_c blocks. When processing block i, the kernel computes its unnormalized attention mass P_i under the running maximum m_i and updates the prefix normalization term l_i. KVFlow records $W_i = P_i/l_i$ (Line 8), the fraction of attention mass assigned to block i within the prefix ending at that block. This adds one scalar per block for the query and head being processed, without exporting token-level attention weights. Because later blocks change the normalization, W_i is not yet the block's final global attention weight.

On-demand Weight Reconstruction. For each recorded snapshot needed by TiDEX, KVFlow converts the prefix ratios into globally normalized block weights:

$$A_i = W_i \prod_{j=i+1}^{T_c} (1 - W_j). \tag{2}$$

Each factor accounts for the reduction in the preceding blocks' normalized mass when a later block is incorporated. The product is empty for the last block, so $A_{T_c} = W_{T_c}$. An efficient reverse scan initializes $R_{T_c+1} = 1$ and, for $i = T_c, \ldots, 1$, computes $A_i = W_i R_{i+1}$ and $R_i = R_{i+1}(1 - W_i)$. The scan takes $O(T_c)$ time and recovers the block-level attention-weight sums under the final softmax normalization. Appendix C provides the derivation, including how Split-KV execution reconstructs attention weights within each split and combines them using per-split log-sum-exp values.

Table 1: Handoff latency (s) and speedup over Llumnix across request-length buckets.

		Multi-News				GovReport			
Model	Metric	4K	8K	12K	16K	8K	16K	24K	32K
Mistral-7B	KVFlow	0.48	0.68	1.05	1.89	0.69	1.47	2.81	4.62
	Llumnix	0.73	1.43	2.12	3.68	1.83	3.38	5.83	8.54
	Speedup	1.52×	2.09×	2.01×	1.95×	2.65×	2.29×	2.08×	1.85×
Qwen2.5-32B	KVFlow	1.20	1.29	1.28	2.17	1.16	1.67	2.02	3.06
	Llumnix	1.90	4.27	8.82	9.80	3.68	7.57	12.33	18.75
	Speedup	1.58×	3.31×	6.89×	4.52×	3.16×	4.54×	6.10×	6.12×

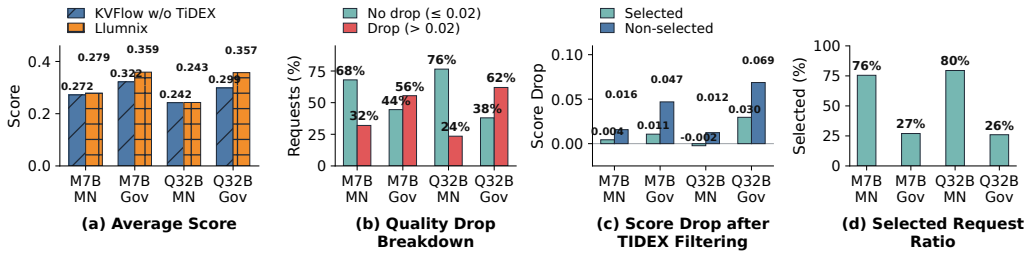


Figure 5: Generation quality and TiDEX filtering across models and datasets. (a) compares the average generation scores of KVFlow without TiDEX and Llumnix. (b) shows the fraction of requests in different quality-drop ranges. (c) reports the mean score drops for requests selected and not selected by TiDEX. (d) reports the ratio of selected requests.

Integration and Overhead. Statistics collection retains the compact snapshots used to compare nearby decoding steps in §4.2; global reconstruction runs when migration is triggered. The block-level signal also fits page-based KV management: in our setting, a statistics block spans several KV pages. Our implementation collects statistics from selected layers, and the overhead evaluation in §5.4 measures both attention-kernel latency and end-to-end TPOT overhead.

5 Evaluation

5.1 Experimental Setup

Implementation and Baselines. We extend Llumnix (Sun et al., 2024) and its vLLM (Kwon et al., 2023) runtime with progressive migration and block-level attention statistics. Baselines are Llumnix and vLLM; all systems use the same vLLM version and initial dispatch policy based on load.

Testbed and Workloads. Our main evaluation uses a 24-GPU NVIDIA RTX 3090 cluster (24 GB per GPU, 100 Gbps RDMA). We evaluate Mistral-7B-v0.3 (Mistral AI, 2024) and Qwen2.5-32B (Qwen, 2024) in FP16, using ShareGPT for conversational requests and Multi-News and GovReport for long-context workloads. Our mixed-workload end-to-end experiments combine ShareGPT and Multi-News. Requests arrive according to a Poisson process with varying QPS; all compared systems use the same request sequence and arrival schedule for each workload and QPS setting. We also conduct supplementary experiments on an eight-GPU NVIDIA A800 server, including mathematical reasoning workloads with long outputs. Memory-footprint and end-to-end serving results are reported in Appendices D.5 and D.6, respectively. Implementation and experimental details are provided in Appendix D.

5.2 Progressive Migration Efficiency

Handoff Latency. We evaluate Mistral-7B-v0.3 and Qwen2.5-32B on two serving instances, using tensor-parallel degrees of 2 and 4, respectively, with requests sampled from Multi-News and GovReport. For these isolated request migrations, we measure handoff latency from the migration trigger until destination-side decoding begins and the source can release the transferred KV cache. Unless otherwise specified, TiDEX uses an attention-weight coverage threshold of $\rho = 0.7$. During

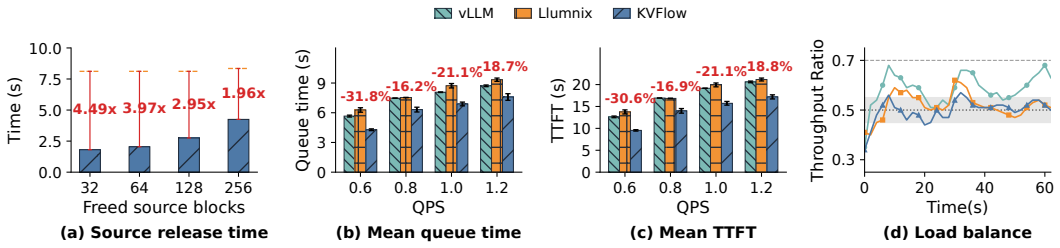


Figure 6: System-level benefits under defragmentation and load imbalance. (a)–(c) show the defragmentation scenario. (d) shows the load-imbalance scenario: throughput ratio over time.

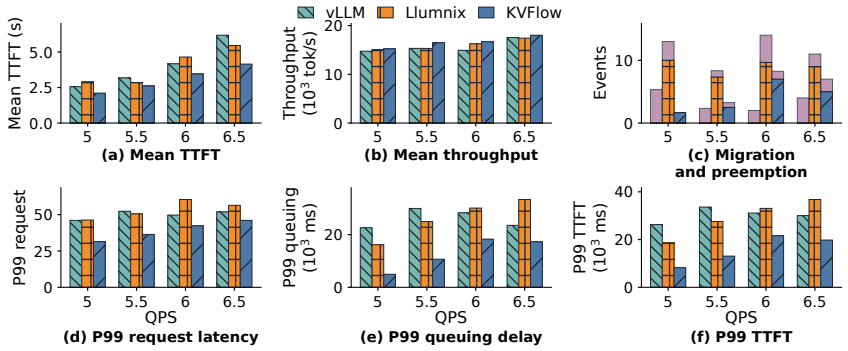


Figure 7: End-to-end performance at different QPS levels.

serving, KV cache migration runs with lower priority than model execution, so handoff latency can exceed the migration times measured in §3.1. Table 1 shows lower handoff latency. The maximum speedups on Multi-News and GovReport are 2.09× and 2.65× for Mistral-7B, and 6.89× and 6.12× for Qwen2.5-32B, respectively. In the 32K GovReport request-length bucket, for example, KVFlow reduces handoff latency for Qwen2.5-32B from 18.75 s to 3.06 s. This reduction reflects earlier handoff, not completion of all KV transfers: Llumnix waits for the full cache, whereas KVFlow continues refill after decoding resumes.

Memory Footprint. We evaluate Qwen2.5-32B on GovReport, with a 10 s observation window per request. KVFlow reduces mean total KV cache occupancy across the source and destination by 29.0% relative to Llumnix, consistent with shortening temporary duplication through incremental source-side cache release. Appendix D.5 provides the detailed results in Table 2, the paired-request aggregation procedure, and supplementary A800 measurements using Gloo and NCCL.

Generation Quality. Without TiDEX filtering, progressive migration can reduce generation quality (Figure 5(a)): the average score decreases from 0.279 to 0.272 for Mistral-7B on Multi-News and from 0.357 to 0.299 for Qwen2.5-32B on GovReport. Figure 5(b) shows that 68% and 38% of requests in these settings, respectively, have a score drop of at most 0.02. These request-level differences show why applying progressive migration indiscriminately can compromise quality even when the average score remains close to the baseline. TiDEX-selected requests exhibit smaller score drops than non-selected requests across workloads (Figure 5(c)); for Mistral-7B on GovReport, the respective average drops are 0.011 and 0.047. Figure 5(d) reports the selected fraction for each model–workload setting, showing that eligible candidates occur across the evaluated workloads. Together, these results support using TiDEX to identify requests less sensitive to partial-cache execution, rather than applying early handoff indiscriminately.

5.3 System-level Benefits

Defragmentation. Figure 6(a) measures the time required to release a target number of source-side KV cache blocks. KVFlow releases these blocks 1.96–4.49× faster than Llumnix across the evaluated target sizes, allowing waiting requests to be admitted earlier. We next consider a serving scenario where one instance becomes highly loaded while the cluster still has sufficient aggregate

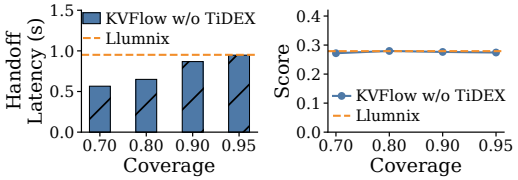


Figure 8: Impact of coverage threshold.

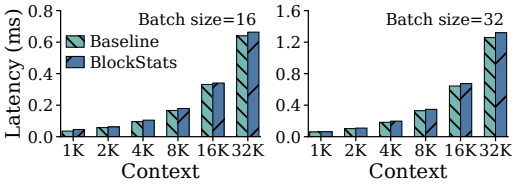


Figure 9: Latency overhead of KVFlow.

memory capacity. Figure 6(b) and Figure 6(c) compare queueing time and TTFT for vLLM, Llumnix, and KVFlow. Across the evaluated QPS settings, KVFlow reduces queueing time by 16.2–31.8% and TTFT by 16.9–30.6% compared to Llumnix.

Load Balancing. We measure the throughput ratio, defined as one instance's share of total output throughput. With two instances, the ideal ratio is 0.5, and we define 0.45–0.55 as the stable region. Following load-imbalance events around 5 s and 35 s, vLLM exhibits sustained throughput skew, while Llumnix adjusts slowly and with larger fluctuations. KVFlow reschedules running requests and keeps the ratio closer to the stable region than either baseline (Figure 6(d)), indicating a faster response to changing load.

End-to-End Serving. We launch 24 Mistral inference instances on the 24-GPU cluster, use NCCL for migration, and issue 500 requests under different QPS settings. Figure 7 shows increasing serving gains under higher load. At 6.5 QPS, KVFlow reduces mean TTFT by 32.9% relative to vLLM and 24.0% relative to Llumnix. Migrating about 1% of requests improves throughput by 3.7% compared with the baselines. The scheduling breakdown (Figure 7(c)) also shows fewer preemptions than both baselines, as migration takes over part of the scheduling burden under memory pressure.

Since migration is triggered for only a small fraction of requests, its effect is concentrated on requests delayed by queueing or preemption. We therefore examine P99 latency to capture these tail benefits. Compared with Llumnix, KVFlow reduces P99 request latency, P99 queueing latency, and P99 TTFT by 27.1%, 53.5%, and 47.3% on average, respectively. Even at 6.5 QPS, when preemption still occurs, earlier memory release allows preempted requests to be re-admitted sooner, shortening their wait for memory and end-to-end latency.

5.4 Deep-Dive

Attention-Weight Coverage. Figure 8 examines the effect of attention-weight coverage on Mistral-7B using 200 Multi-News requests. We disable TiDEX-based request filtering and trigger migration at approximately the same decoding step, while retaining attention-based critical-block selection. Reducing coverage from 95% to 70% decreases the average time spent in the first two migration stages from 0.95 s to 0.57 s. Meanwhile, mean ROUGE-L remains between 0.272 and 0.280, compared with 0.279 for Llumnix. These results support using a lower coverage threshold to reduce pre-handoff work with limited average quality loss in this setting.

System Overhead. Using the Mistral-7B-v0.3 model configuration, we benchmark vanilla FlashAttention and the BlockStats extension across batch sizes and context lengths. The extension adds an average of 0.014 ms per attention computation (Figure 9). We also measure TPOT overhead for context lengths of 1K–32K tokens; it is approximately 0.33 ms at 32K. Collecting block-level statistics only from selected layers limits the additional work on the decoding path, keeping the runtime overhead small.

6 Conclusion

In this paper, we present KVFlow, an LLM inference rescheduling system that improves the responsiveness of runtime request migration by decoupling request handoff from full KV cache transfer. KVFlow exploits temporal sparsity stability to identify stable critical KV cache blocks for early destination-side decoding, while progressively refilling the remaining KV cache and releasing source-side memory during migration. Our evaluation shows that KVFlow improves migration efficiency and delivers better serving performance.

AI USE STATEMENT

We used generative AI tools primarily for language polishing and assistance with figure preparation, including improvements to wording, clarity, and visual presentation. These tools also assisted with drafting and revising portions of the manuscript, manuscript organization, and table and LaTeX editing. We additionally used them to review algorithmic notation and mathematical explanations, provide feedback on experimental methodology, and assist with interpreting experimental results. Their use in reviewing mathematical content was limited to notation and exposition; they were not used to construct or prove mathematical claims. The authors are responsible for verifying AI-assisted content and retain full responsibility for the final manuscript, including its methods, experimental results, and conclusions.

REFERENCES

Saurabh Agarwal, Anyong Mao, Aditya Akella, and Shivaram Venkataraman. Symphony: Improving memory management for llm inference workloads, 2024. URL https://arxiv.org/abs/2412.16434.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 3119–3137. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.172. URL https://doi.org/10.18653/v1/2024.acl-long.172.

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pp. 273–286, USA, 2005. USENIX Association.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

DeepSeek-AI. Deepseek-v3 technical report. *CoRR*, abs/2412.19437, 2024. doi: 10.48550/ARXIV.2412.19437. URL https://doi.org/10.48550/arXiv.2412.19437.

Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in llm serving with hcache. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 128–143, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696072. URL https://doi.org/10.1145/3689031.3696072.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/be83ab3ecd0db773eb2dc1b0a17836a1-Paper-round2.pdf.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515, 2024.

Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, Ana Klimovic, and Eiko Yoneki. Thunderserve: High-performance and cost-efficient llm serving in cloud environments, 2025. URL https://arxiv.org/abs/2502.09334.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. Infinigen: Efficient generative inference of large language models with dynamic KV cache management. In Ada Gavrilovska and Douglas B. Terry (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 155–172. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/lee.

Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. In *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. mlsys.org, 2025.

Meta Llama. Llama-3.3-70b-instruct. https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct, 2024. Hugging Face model card. Accessed: 2026-04-28.

Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (eds.), *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pp. 1112–1127. ACM, 2024. doi: 10.1145/3620665.3640411. URL https://doi.org/10.1145/3620665.3640411.

Mistral AI. Mistral-7b-instruct-v0.3. https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3, 2024. Accessed: 2026-04-29.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 155–170, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Qwen. Qwen2.5: A party of foundation models!, September 2024. URL https://qwen.ai/blog?id=qwen2.5.

QwenTeam. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

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

StepFun, :, Bin Wang, Bojun Wang, Changyi Wan, Guanzhe Huang, Hanpeng Hu, Haonan Jia, Hao Nie, Mingliang Li, Nuo Chen, Siyu Chen, Song Yuan, Wuxun Xie, Xiaoniu Song, Xing Chen, Xingping Yang, Xuelin Zhang, Yanbo Yu, Yaoyu Wang, Yibo Zhu, Yimin Jiang, Yu Zhou, Yuanwei Lu, Houyi Li, Jingcheng Hu, Ka Man Lo, Ailin Huang, Binxing Jiao, Bo Li, Boyu Chen, Changxin Miao, Chang Lou, Chen Hu, Chen Xu, Chenfeng Yu, Chengyuan Yao, Daokuan Lv,

Dapeng Shi, Deshan Sun, Ding Huang, Dingyuan Hu, Dongqing Pang, Enle Liu, Fajie Zhang, Fanqi Wan, Gulin Yan, Han Zhang, Han Zhou, Hanghao Wu, Hangyu Guo, Hanqi Chen, Hanshan Zhang, Hao Wu, Haocheng Zhang, Haolong Yan, Haoran Lv, Haoran Wei, Hebin Zhou, Heng Wang, Heng Wang, Hongxin Li, Hongyu Zhou, Hongyuan Wang, Huiyong Guo, Jia Wang, Jiahao Gong, Jialing Xie, Jian Zhou, Jianjian Sun, Jiaoren Wu, Jiaran Zhang, Jiayu Liu, Jie Cheng, Jie Luo, Jie Yan, Jie Yang, Jieyi Hou, Jinguang Zhang, Jinlan Cao, Jisheng Yin, Junfeng Liu, Junhao Huang, Junzhe Lin, Kaijun Tan, Kaixiang Li, Kang An, Kangheng Lin, Kenkun Liu, Lei Yang, Liang Zhao, Liangyu Chen, Lieyu Shi, Liguo Tan, Lin Lin, Lin Zhang, Lina Chen, Liwen Huang, Liying Shi, Longlong Gu, Mei Chen, Mengqiang Ren, Ming Li, Mingzhe Chen, Na Wang, Nan Wu, Qi Han, Qian Zhao, Qiang Zhang, Qianni Liu, Qiaohui Chen, Qiling Wu, Qinglin He, Qinyuan Tan, Qiufeng Wang, Qiuping Wu, Qiuyan Liang, Quan Sun, Rui Li, Ruihang Miao, Ruosi Wan, Ruyan Guo, Shangwu Zhong, Shaoliang Pang, Shengjie Fan, Shijie Shang, Shilei Jiang, Shiliang Yang, Shiming Hao, Shuli Gao, Siming Huang, Siqi Liu, Tiancheng Cao, Tianhao Cheng, Tianhao Peng, Wang You, Wei Ji, Wen Sun, Wenjin Deng, Wenqing He, Wenzhen Zheng, Xi Chen, Xiangwen Kong, Xianzhen Luo, Xiaobo Yang, Xiaojia Liu, Xiaoxiao Ren, Xin Han, Xin Li, Xin Wu, Xu Zhao, Yanan Wei, Yang Li, Yangguang Li, Yangshijie Xu, Yanming Xu, Yaqiang Shi, Yeqing Shen, Yi Yang, Yifei Yang, Yifeng Gong, Yihan Chen, Yijing Yang, Yinmin Zhang, Yizhuang Zhou, Yuanhao Ding, Yuantao Fan, Yuanzhen Yang, Yuchu Luo, Yue Peng, Yufan Lu, Yuhang Deng, Yuhe Yin, Yujie Liu, Yukun Chen, Yuling Zhao, Yun Mou, Yunlong Li, Yunzhou Ju, Yusheng Li, Yuxiang Yang, Yuxiang Zhang, Yuyang Chen, Zejia Weng, Zhe Xie, Zheng Ge, Zheng Gong, Zhenyi Lu, Zhewei Huang, Zhichao Chang, Zhiguo Huang, Zhirui Wang, Zidong Yang, Zili Wang, Ziqi Wang, Zixin Zhang, Binxing Jiao, Daxin Jiang, Heung-Yeung Shum, and Xiangyu Zhang. Step-3 is large yet affordable: Model-system co-design for cost-effective decoding, 2025. URL https://arxiv.org/abs/2507.19427.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In Ada Gavrilovska and Douglas B. Terry (eds.), *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pp. 173–191. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/sun-biao.

Hanshi Sun, Li-Wen Chang, Wenlei Bao, Size Zheng, Ningxin Zheng, Xin Liu, Harry Dong, Yuejie Chi, and Beidi Chen. Shadowkv: KV cache in shadows for high-throughput long-context LLM inference. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/sun25b.html.

Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards end-to-end optimization of llm-based applications with ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, pp. 1302–1316, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. doi: 10.1145/3676641.3716278. URL https://doi.org/10.1145/3676641.3716278.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. QUEST: query-aware sparsity for efficient long-context LLM inference. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=KzACYw0MTV.

Yi Xu, Ziming Mao, Xiangxi Mo, Shu Liu, and Ion Stoica. Pie: Pooling cpu memory for llm inference, 2024. URL https://arxiv.org/abs/2411.09317.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving. In M. Zaharia, G. Joshi, and Y. Lin (eds.), *Proceedings of Machine Learning and Systems*, volume 7. MLSys, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/file/dbf02b21d77409a2db30e56866a8ab3a-Paper-Conference.pdf.

Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, Ion Stoica, Harry Xu, and Ying Sheng. Prism: Unleashing gpu sharing for cost-efficient multi-llm serving, 2025. URL https://arxiv.org/abs/2505.04021.

Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. BlitzScale: Fast and live large model autoscaling with o(1) host caching. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 275–293, Boston, MA, July 2025. USENIX Association. ISBN 978-1-939133-47-2. URL https://www.usenix.org/conference/osdi25/presentation/zhang-dingyan.

Youpeng Zhao, Di Wu, and Jun Wang. Alisa: Accelerating large language model inference via sparsity-aware kv caching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 1005–1017. IEEE, 2024.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

APPENDIX

A SYSTEM ARCHITECTURE AND PROGRESSIVE MIGRATION PROTOCOL

This appendix complements §4 with the component organization and state-management details of KVFlow. The main text describes candidate selection and the three migration stages; here we explain how the serving runtime implements them.

A.1 SYSTEM ARCHITECTURE

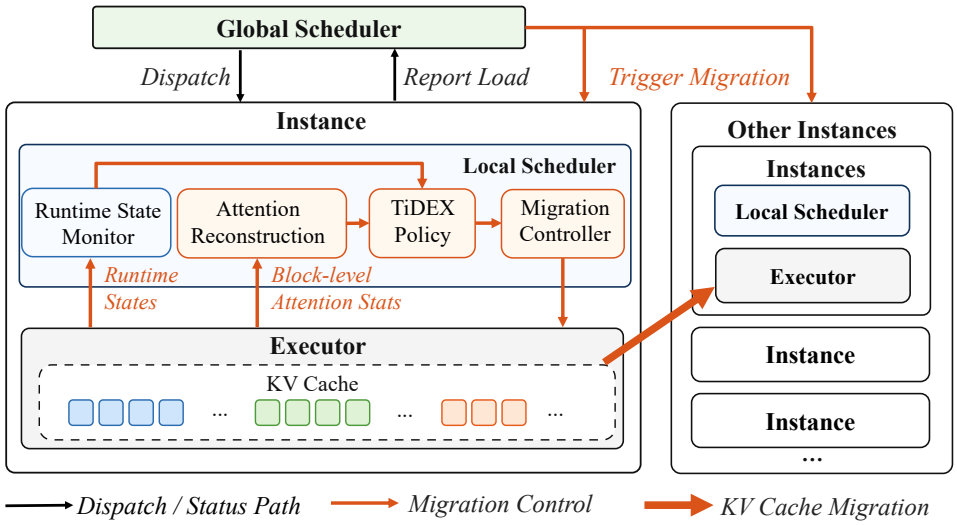


Figure 10: KVFlow architecture. The global scheduler coordinates dispatch and migration across instances; local attention statistics and runtime state guide TiDEX, and the migration controller executes progressive KV cache migration.

Figure 10 shows KVFlow's organization on top of Llumnix (Sun et al., 2024) and a modified vLLM (Kwon et al., 2023) runtime. A global scheduler dispatches incoming requests and identifies migration source–destination pairs using per-instance load and available KV cache capacity. Migration is triggered to address conditions such as load imbalance or memory fragmentation.

Within each instance, the executor runs inference and manages the KV cache. The attention-kernel extension exposes compact block-level statistics, from which the local scheduler reconstructs attention weights when migration is triggered. The local scheduler combines these statistics with runtime state and uses TiDEX to select a migration candidate and its critical KV subset. The migration controller then coordinates transfer and request handoff with the destination. This organization connects attention-statistics collection, candidate selection, and progressive migration.

A.2 MIGRATION PROTOCOL AND STATE MANAGEMENT

Figure 11 illustrates the three stages described in §4.1. The coordinator reads the request's block tables and transfers the selected KV blocks through the migration backend using Gloo or NCCL with a configurable transfer buffer. The following operations support destination-side decoding before the full cache arrives.

Critical transfer and handoff. The destination pre-allocates storage for the critical KV blocks, which the source transfers while continuing inference. To hand off the request, the source pauses execution and transfers the KV cache generated during this initial transfer, together with the request metadata. The destination installs the received blocks in its block tables and takes over decoding only after the critical subset, incremental KV cache, and metadata are available. At this point the source stops executing the request and may release the blocks already incorporated at the destination, while retaining the remaining blocks for refill.

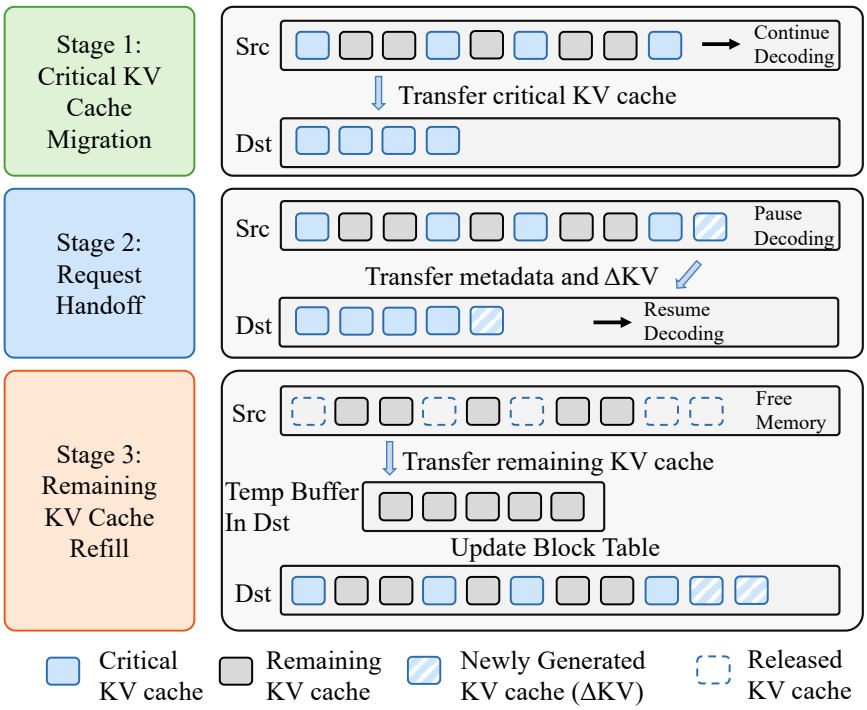


Figure 11: Progressive migration protocol: critical KV transfer, request handoff, and asynchronous refill of the remaining KV cache. Decoding moves to the destination at handoff, and the source releases blocks as they are incorporated at the destination.

Asynchronous refill. Remaining KV blocks may arrive out of sequence order after the destination has advanced the request. The destination therefore receives each refill batch into temporary KV cache storage and temporary block tables. At an engine-step boundary, it merges the batch into the running request's block table, mapping the received physical blocks to their original logical block indices. PagedAttention's separation of logical indices from physical storage allows these blocks to be restored to their original positions rather than appended as newly generated KV cache.

Partial source-side release. After a refill batch has been incorporated at the destination, the source releases the corresponding blocks. Since partial release can change the source-side block-table layout, the coordinator maintains logical-to-physical mappings throughout migration so that subsequent transfers and releases still address the intended blocks. The source retains blocks that have not yet been incorporated at the destination.

These operations maintain the association between KV blocks and their logical positions while coordinating execution ownership. As discussed in the main text, decoding during refill uses only the available KV cache and is not guaranteed to reproduce full-KV outputs; generation quality is evaluated separately from protocol state consistency.

B Details of TiDEX Request Selection

TiDEX evaluates migration candidates along two dimensions: the useful source-side memory relief relative to the initial transfer cost, and the temporal stability of the critical KV subset. This appendix illustrates these two parts of the selection process described in §4.2.

B.1 Migration Benefit Estimation

Figure 12 illustrates the trade-off between source-side memory relief and initial migration cost. The left panel decomposes each request's KV cache into critical and remaining subsets. The critical-subset size serves as a proxy for pre-handoff transfer cost, while the total KV cache size indicates

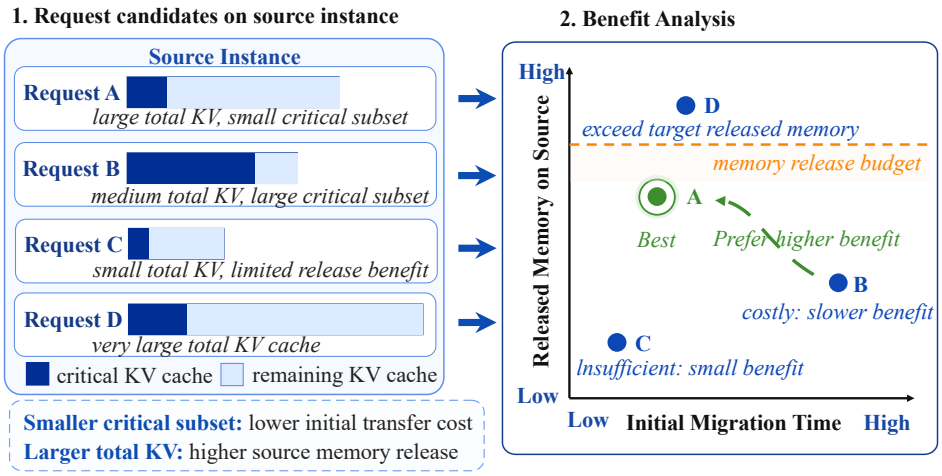


Figure 12: Illustration of request selection based on benefits analysis.

the potential memory relief from migrating the request. This relief is realized progressively as the source releases transferred blocks; it is not all available at handoff. The right panel compares these two quantities qualitatively; the dashed line marks the source's target memory release.

Request A illustrates a favorable combination of substantial potential memory relief and a small critical subset. Request B requires a larger initial transfer, whereas request C offers limited memory relief despite its small critical subset. Request D can release more memory than the current target, but Equation 1 caps useful relief at that target. The figure illustrates the benefit trade-off; assessing temporal stability requires the additional comparison below.

B.2 Temporal Stability Assessment

A favorable benefit score alone does not establish whether the critical KV subset remains useful during partial-KV decoding. TiDEX therefore compares reconstructed block-level attention weights at the current decoding step, A_T, and at an earlier step, A_{T-N}. These weights are obtained using the attention-statistics extension in §4.3. Figure 13 illustrates the comparison.

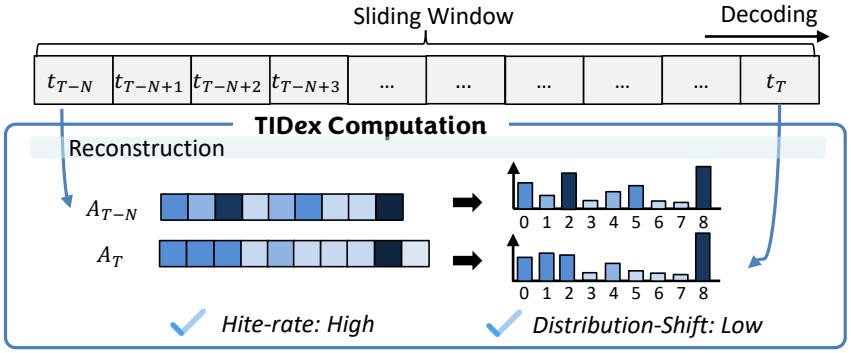


Figure 13: Illustration of temporal stability assessment in TiDEX. Reconstructed block-level attention weights at steps $T-N$ and T are compared using top-K block overlap and attention-distribution shift.

Critical-block overlap. For a candidate request r, let $K = K_\rho(r)$ denote the critical-subset size selected at the current step. Let S_T and S_{T-N} denote the sets of top-K block identities at the two steps. The top-K hit rate measures the fraction of selected blocks shared by the two sets:

$$\mathrm{HitRate}(r) = \frac{|S_T \cap S_{T-N}|}{K}. \tag{3}$$

A high hit rate indicates that the dominant blocks remain similar across the comparison interval.

Attention-distribution shift. Set overlap does not describe how attention mass changes among blocks. TiDEX complements the hit rate with JS-shift, the Jensen–Shannon divergence between the normalized block-level attention distributions at the two steps. A low JS-shift indicates a small change in the distribution of attention mass. In Figure 13, the high overlap and low distribution shift illustrate a temporally stable candidate.

Stability criterion. In the evaluated configuration, the snapshots are 16 decoding iterations apart, and a candidate passes the stability check when its top-K hit rate is at least 0.60 and its JS-shift is at most 3%. These are the same settings summarized in Appendix D.4. The benefit score measures the value of migrating a request, while the stability check assesses whether its critical subset is suitable for decoding before full KV refill.

C Derivation of Block-level Statistics

This appendix derives how KVFlow converts the prefix-normalized ratios recorded by the attention kernel into globally normalized block-level attention-weight sums. We first consider the regular non-split execution and then extend the derivation to split-KV execution.

C.1 From Prefix-normalized Ratios to Global Block-level Weights

Consider a single query and one attention head. Let the KV cache be divided into T_c blocks, indexed by $i = 1, \ldots, T_c$ from the earliest to the latest block. For block i, let M_i denote its unnormalized attention-weight sum under the final safe-softmax semantics, i.e.,

$$M_i = \sum_{t \in \mathcal{B}_i} \exp(s_t - m),$$

where $\mathcal{B}_i$ is the set of tokens in block i, s_t is the pre-softmax logit of token t, and m is the maximum logit over the entire context. The final normalized attention-weight sum assigned to block i is:

$$A_i = \frac{M_i}{\sum_{k=1}^{T_c} M_k}.$$

During online softmax, after the kernel processes block i, the normalization term corresponds to the prefix ending at block i. KVFlow records:

$$W_i = \frac{M_i}{\sum_{k=1}^{i} M_k}.$$

Thus, W_i is not directly the final global attention-weight sum. Instead, it is the fraction of the current prefix attention contributed by block i.

We now show how to recover A_i from the recorded ratios W_i. By definition,

$$1 - W_j = 1 - \frac{M_j}{\sum_{k=1}^{j} M_k} = \frac{\sum_{k=1}^{j-1} M_k}{\sum_{k=1}^{j} M_k}.$$

Therefore, for any block i,

$$\prod_{j=i+1}^{T_c} (1 - W_j) = \prod_{j=i+1}^{T_c} \frac{\sum_{k=1}^{j-1} M_k}{\sum_{k=1}^{j} M_k}.$$

This product telescopes:

$$\prod_{j=i+1}^{T_c} (1 - W_j) = \frac{\sum_{k=1}^{i} M_k}{\sum_{k=1}^{T_c} M_k}.$$

Multiplying this suffix product by W_i gives:

$$W_i \prod_{j=i+1}^{T_c} (1 - W_j) = \frac{M_i}{\sum_{k=1}^{i} M_k} \cdot \frac{\sum_{k=1}^{i} M_k}{\sum_{k=1}^{T_c} M_k} = \frac{M_i}{\sum_{k=1}^{T_c} M_k}.$$

Hence,

$$A_i = W_i \prod_{j=i+1}^{T_c} (1 - W_j).$$

For $i = T_c$, the product is empty and equals one; hence $A_{T_c} = W_{T_c}$. KVFlow implements this reconstruction using a reverse suffix-product scan from $i = T_c$ down to $i = 1$. Let

$$R_{T_c+1} = 1, \qquad R_i = R_{i+1}(1 - W_i).$$

Then,

$$R_{i+1} = \prod_{j=i+1}^{T_c} (1 - W_j),$$

and therefore:

$$A_i = W_i R_{i+1}.$$

This scan visits each block once and incurs $O(T_c)$ post-processing cost.

C.2 Extension to Split-KV Execution

In split-KV execution, the KV context is partitioned into multiple splits. Let s index a split, and let T_s denote the number of blocks in split s, indexed by $i = 1, \ldots, T_s$. For clarity, we define the unnormalized attention-weight sum of block i in split s as $M_{s,i} = \sum_{t \in \mathcal{B}_{s,i}} \exp(s_t)$ using unshifted logits, where $\mathcal{B}_{s,i}$ denotes the tokens in that block. In practice, the kernel uses max-shifted exponentials for numerical stability and includes the shift when computing each split's log-sum-exp. The global block-level attention-weight sum should be:

$$A_{s,i} = \frac{M_{s,i}}{\sum_{s'} \sum_{k=1}^{T_{s'}} M_{s',k}}.$$

Within each split, the attention kernel performs online softmax over only the KV tokens in that split. Therefore, KVFlow first applies the same suffix-product reconstruction inside each split:

$$\tilde{A}_{s,i} = W_{s,i} \prod_{j=i+1}^{T_s} (1 - W_{s,j}),$$

where $\tilde{A}_{s,i}$ is normalized only within split s:

$$\tilde{A}_{s,i} = \frac{M_{s,i}}{\sum_{k=1}^{T_s} M_{s,k}}.$$

To make block weights comparable across splits, we need to reweight each split by its contribution to the global softmax normalization. Let

$$Z_s = \sum_{k=1}^{T_s} M_{s,k}$$

be the unnormalized attention-weight sum of split s. The global weight of split s is:

$$\alpha_s = \frac{Z_s}{\sum_{s'} Z_{s'}}.$$

FlashAttention exposes this information through the per-split log-sum-exp value:

$$\mathrm{LSE}_s = \log Z_s.$$

Thus,

$$\alpha_s = \frac{\exp(\mathrm{LSE}_s)}{\sum_{s'} \exp(\mathrm{LSE}_{s'})}.$$

Finally, the globally normalized attention-weight sum of block i in split s is:

$$A_{s,i} = \alpha_s \tilde{A}_{s,i}.$$

Substituting the within-split reconstruction gives:

$$A_{s,i} = \alpha_s W_{s,i} \prod_{j=i+1}^{T_s} (1 - W_{s,j}).$$

Equivalently, initialize $R_{s,T_s+1} = 1$ and scan from $i = T_s$ down to $i = 1$ using $R_{s,i} = R_{s,i+1}(1 - W_{s,i})$. The within-split suffix factor is

$$R_{s,i+1} = \prod_{j=i+1}^{T_s} (1 - W_{s,j}),$$

and we obtain:

$$A_{s,i} = \alpha_s W_{s,i} R_{s,i+1}.$$

This derivation shows that KVFlow can recover globally comparable block-level attention-weight sums under both regular and split-KV execution. The split-KV case only requires an additional reweighting step using the per-split log-sum-exp values already produced by the attention kernel.

D Implementation and Experimental Details

This appendix provides implementation and configuration details for the evaluation in §5.1 and the motivation experiments in §3. The system architecture and migration protocol are described in Appendix A.

D.1 Motivation Experiment Settings

Live-migration completion time. For Figure 2, we measure Llumnix live migration using two serving instances deployed on two 8-GPU servers connected through a 100 Gbps network. We evaluate Mistral-7B (Mistral AI, 2024), Qwen2.5-32B (Qwen, 2024), and Llama-3.3-70B (Meta Llama, 2024) across different context lengths. The hardware configuration is described in Appendix D.3.

Partial-cache decoding and refill. For the stability and generation-quality experiments in §3.2, inference with full KV cache serves as the baseline. We emulate progressive migration by allowing the destination to decode with selected KV pages before restoring the full cache. Unless otherwise specified, we retain the top 25% of KV pages selected by page-level attention weights. We evaluate Mistral-7B on GovReport and Multi-News using ROUGE-L (Bai et al., 2024), and Qwen3-4B-Thinking (QwenTeam, 2025) on MATH-500 (Hendrycks et al., 2021) using accuracy. We measure cosine similarity between partial-cache and full-cache attention distributions across decoding steps, and vary the step at which the omitted cache is refilled to evaluate generation quality.

Request-dependent critical-subset size. For Figure 4(c), we vary the retained KV ratio rather than fixing it at 25%. At the migration point, we measure the minimum KV cache ratio required by each request to reach attention-weight coverage targets of 90%, 92.5%, 95%, and 97.5%. For each dataset and target, the main text compares the requests with the highest and lowest required ratios. These page-level motivation experiments are distinct from the block-level TiDEX configuration and coverage experiment described in Appendix D.4.

D.2 Implementation Details

We implement KVFlow on top of Llumnix and a modified vLLM runtime, using the same vLLM version as Llumnix for the comparison. We extend Llumnix's migration coordinator and vLLM's scheduler/block manager to support handoff with partial KV cache, asynchronous refill, and partial source-side release. KV blocks are transferred through Gloo or NCCL. Appendix A.2 describes the transfer buffers, block-table updates, and logical-to-physical mappings used during migration.

We also extend vLLM's FlashAttention decoding path to return compact statistics alongside normal decoding outputs. These statistics include block-level attention ratios and split-level log-sum-exp values, which support the reconstruction in §4.3 without materializing token-level attention weights.

D.3 Evaluation Configuration

The main testbed contains 24 NVIDIA RTX 3090 GPUs with 24 GB memory per GPU and 100 Gbps RDMA connectivity. The second testbed is a single server with eight NVIDIA A800-SXM4 GPUs (80 GB per GPU), NVLink connectivity, and four 200 Gbps InfiniBand ports supporting RDMA. The memory-footprint experiments use two GPUs in this server, with one serving instance per GPU, and evaluate both Gloo and NCCL. The evaluation uses Mistral-7B-v0.3, Qwen2.5-32B, and DeepSeek-R1-Distill-Qwen-14B in FP16. GPUs are partitioned into serving instances according to the experiment's tensor-parallel configuration. Gloo and NCCL are used as migration backends unless otherwise specified.

For the handoff-latency experiments in §5.2, each model runs on two inference instances, with tensor-parallel degrees of 2 for Mistral-7B-v0.3 and 4 for Qwen2.5-32B. KV transfer has lower priority than model execution. The mixed-workload end-to-end experiment uses 24 Mistral inference instances on the 24-GPU cluster, with NCCL as the migration backend. The compared systems share the initial dispatch policy described in the main text.

D.4 Evaluation Workloads and Parameters

ShareGPT supplies conversational requests, while Multi-News and GovReport supply long-context requests. The mixed-workload end-to-end experiment combines ShareGPT and Multi-News to obtain requests of different lengths. Following prior work (Sun et al., 2024; Zhong et al., 2024), request arrivals follow a Poisson process, and we vary QPS to control offered load. The mixed-workload end-to-end experiment issues 500 requests under different QPS settings. For each workload and QPS setting, all compared systems use the same request sequence and arrival schedule. The handoff-latency experiments sample running requests from Multi-News and GovReport; the memory-footprint experiments use GovReport with Qwen2.5-32B, with the paired-sample counts and aggregation procedure described in Appendix D.5.

Mathematical Reasoning Workload. The DeepSeek-R1-Distill-Qwen-14B experiments use eight replicas on the A800 server with tensor parallelism of one, each allowing up to 64 concurrent sequences and setting the GPU memory utilization parameter to 0.60. CUDA Graph execution is enabled, and output processing is synchronous in all compared systems. Each run contains 500 MATH-500 requests, with generation terminating naturally or upon reaching the maximum output length of 16,384 tokens.

Unless otherwise specified, the critical KV subset is selected with attention-weight coverage $\rho = 0.7$. Across all evaluated models and datasets, a candidate must have a top-K hit rate of at least 0.60 and JS-shift of at most 3%, using attention snapshots 16 decoding iterations apart. Appendix B.2 explains these metrics, and the coverage study in §5.4 evaluates the coverage trade-off. That coverage experiment disables TiDEX-based request filtering to isolate the effect of ρ.

D.5 Per-request Migration Memory Footprint

Table 2 summarizes KV cache occupancy for Qwen2.5-32B on GovReport using the same 10 s per-request observation window across the A800 and RTX 3090 experiments. In these experiments, the reductions are computed as $(M_{\mathrm{Llumnix}} - M_{\mathrm{KVFlow}})/M_{\mathrm{Llumnix}}$ for each reported metric.

Table 2: Mean KV cache occupancy for Qwen2.5-32B on GovReport across GPU platforms and migration backends. Values are in KV blocks; reductions are relative to Llumnix.

GPU / Backend	Metric	KVFlow	Llumnix	Reduction
RTX 3090 / Gloo	Mean source	401.51	530.98	24.38%
	Mean destination	445.05	661.28	32.70%
	Mean total	846.55	1192.26	29.00%
A800 / Gloo	Mean source	245.38	364.90	32.75%
	Mean destination	504.90	643.07	21.49%
	Mean total	750.28	1007.98	25.57%
A800 / NCCL	Mean source	279.57	487.60	42.66%
	Mean destination	482.47	642.37	24.89%
	Mean total	762.04	1129.97	32.56%

Additional per-request visualization. Figure 14 shows the earlier 50-request RTX 3090 experiment with Qwen2.5-32B on GovReport. The panels report occupied KV cache blocks at the source, at the destination, and across both instances. These curves illustrate request-level variation in a separate experiment; they are not the per-request data underlying the expanded summary in Table 2.

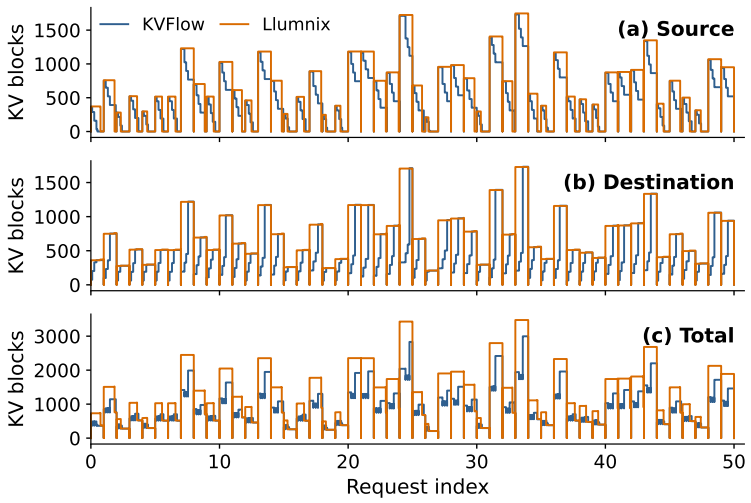


Figure 14: Per-request KV cache memory footprint for Qwen2.5-32B on 50 GovReport requests in the earlier RTX 3090 experiment. The panels show source, destination, and total occupied KV cache blocks.

A800 per-request results. Figure 15 and Figure 16 extend this visualization to Gloo and NCCL on a single server with two A800 GPUs, using one serving instance per GPU. Each plot shows 50 requests successfully matched between KVFlow and Llumnix for Qwen2.5-32B on GovReport. These traces illustrate source-side release, destination-side cache accumulation, and their combined memory footprint during migration. The A800 aggregate results in Table 2 use all 194 jointly successful pairs per backend, rather than only the 50 requests visualized here.

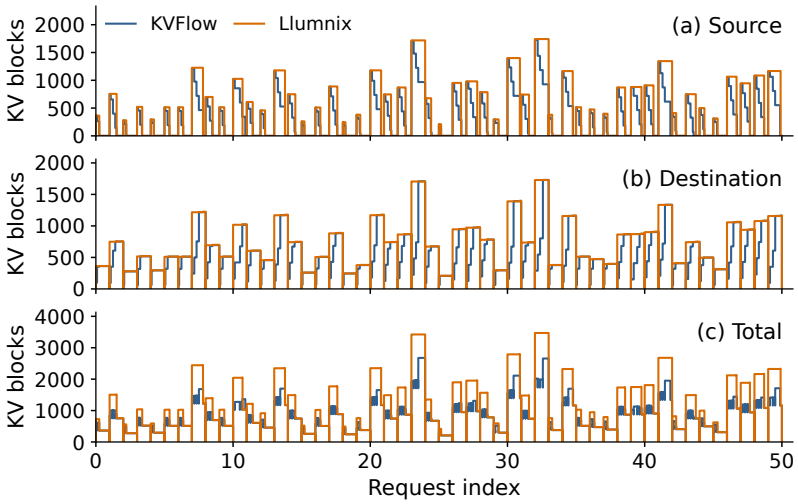


Figure 15: Per-request KV cache memory footprint for Qwen2.5-32B on 50 matched GovReport requests using Gloo on two A800 GPUs within one server. Panels (a)–(c) show occupied KV cache blocks at the source, at the destination, and across both instances, respectively.

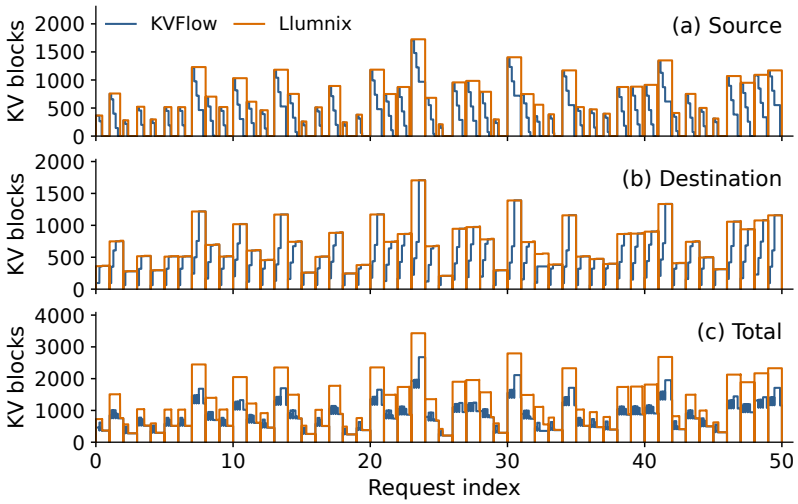


Figure 16: Per-request KV cache memory footprint for Qwen2.5-32B on 50 matched GovReport requests using NCCL on two A800 GPUs within one server. Panels (a)–(c) show occupied KV cache blocks at the source, at the destination, and across both instances, respectively.

D.6 Supplementary End-to-End Evaluation on A800

We evaluate DeepSeek-R1-Distill-Qwen-14B in FP16 on MATH-500 using a single server with eight NVIDIA A800-SXM4 GPUs (80 GB each) connected via NVLink. We deploy eight single-GPU serving instances and compare vLLM, Llumnix, and KVFlow at 4, 5, 6, and 7 QPS. Each run contains 500 requests. Requests follow a Poisson arrival process, with the same request sequence and arrival schedule across systems at each QPS. Results are averaged over the available runs for each system/QPS setting; the number of runs varies across settings. Runtime and generation settings are provided in Appendix D.4.

Table 3: End-to-end serving results for DeepSeek-R1-Distill-Qwen-14B on MATH-500 using eight A800 GPUs. Values are averaged over the available runs for each system/QPS setting, with 500 requests per run. TTFT is in seconds; successful migrations and preemptions are mean counts per run.

QPS	System	P99 TTFT	Mean TTFT	Successful migrations	Preemptions
4	vLLM	3.994	0.726	0.00	40.00
	Llumnix	9.380	1.542	24.67	0.33
	KVFlow	1.275	0.224	24.00	0.00
5	vLLM	5.130	0.737	0.00	33.50
	Llumnix	5.183	0.920	21.00	0.00
	KVFlow	2.276	0.342	19.50	0.00
6	vLLM	7.118	1.308	0.00	94.00
	Llumnix	5.928	1.027	34.50	3.00
	KVFlow	0.903	0.212	29.50	1.00
7	vLLM	4.868	0.782	0.00	55.00
	Llumnix	2.105	0.404	23.00	1.00
	KVFlow	1.055	0.236	35.00	0.00

Table 3 shows that KVFlow lowers both mean and P99 TTFT for this long-output mathematical reasoning workload at every evaluated QPS. Based on the reported averages, KVFlow reduces P99 TTFT by 55.6–87.3% compared with vLLM and 49.9–86.4% compared with Llumnix; the corresponding mean TTFT reductions are 53.6–83.8% and 41.6–85.5%.

Long generations continue to accumulate KV cache, which can sustain memory pressure and increase the risk of preemption. In these runs, vLLM averages 33.50–94.00 preemptions per run, whereas KVFlow averages 0, 0, 1, and 0 at 4, 5, 6, and 7 QPS, respectively. Llumnix averages 0.33, 0, 3.00, and 1.00 at the same QPS settings. Thus, KVFlow reduces preemptions compared with vLLM at every evaluated QPS and incurs no more preemptions than Llumnix. At 6 QPS, for example, KVFlow averages one preemption per run, compared with 94 for vLLM and three for Llumnix, while also achieving lower mean and P99 TTFT than both baselines.

Together, these results show that KVFlow improves serving latency while limiting preemption in this long-output workload. The results are consistent with progressive migration's earlier source-side memory release easing request-admission pressure and reducing reliance on preemption. These A800 measurements complement the main mixed-workload evaluation.

E Related Work and Discussion

E.1 Related Work

Inference Serving: vLLM introduces PagedAttention (Kwon et al., 2023), which manages the KV cache in pages to effectively reduce memory fragmentation on a single machine. Multi-instance serving additionally requires efficient allocation of resources across instances. SYMPHONY (Agarwal et al., 2024) migrates KV caches across turns for multi-turn workloads. DistServe separates the prefill and decoding phases to reduce interference and improve goodput under latency constraints (Zhong et al., 2024). Step-3 decouples Attention and Mixture-of-Experts (MoE) computations across multiple instances (StepFun et al., 2025). Llumnix (Sun et al., 2024) supports runtime rescheduling through live migration, continuing source-side decoding during transfer but delaying handoff until the full KV cache is available. KVFlow focuses on this migration interval: it hands off decoding with a critical subset and incrementally releases source-side memory as the destination incorporates the remaining cache.

Sparsity: KV cache sparsity, compression, and offloading reduce the memory and data-access costs of LLM inference. FlexGen (Sheng et al., 2023) targets throughput-oriented generative inference by combining offloading with compression of model weights and the KV cache. Alisa leverages sparsity in large models to tackle the challenges posed by oversized KV cache during single-instance

inference (Zhao et al., 2024). InfiniGen dynamically prefetches only important KV cache blocks, reducing fetch overhead and thereby enhancing overall performance (Lee et al., 2024). QUEST (Tang et al., 2024) uses query-aware KV selection to reduce attention computation, while ShadowKV (Sun et al., 2025) combines a low-rank key cache with sparse retrieval of offloaded KV cache. For progressive migration, the critical subset serves a different purpose: migration spans a short decoding window, during which the initially migrated KV cache must remain useful until refill completes. KVFlow therefore uses temporal sparsity stability to select requests and critical subsets for early handoff, with background refill restoring the remaining cache. The goal is timely source-side resource relief during runtime rescheduling.

E.2 Discussion

KVFlow provides progressive migration as a runtime primitive for moving active LLM requests across serving instances. This primitive can support other scenarios.

RL Rollout. RL rollout workloads (Sheng et al., 2024; Mei et al., 2025) can suffer from imbalance because some workers finish early while others continue serving long generations with large KV cache states. Progressive KV cache migration offers a complementary mechanism to asynchronous execution: it can move ongoing rollout requests across workers, rebalancing active generations without waiting for them to complete. This makes KVFlow potentially useful for improving rollout efficiency.

Decode-to-Prefill Reconfiguration. KVFlow can also support dynamic resource reconfiguration in prefill–decode disaggregated serving. When traffic shifts, a decode instance may need to be converted into a prefill instance, but active decode requests and their KV cache states can delay this transition. Progressive migration can quickly hand off these requests to other decode instances.