KVFLOW: CROSS-INSTANCE KV CACHE FLOW FOR EFFICIENT AND BALANCED LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-instance large language model (LLM) serving must balance KV cache reuse and load balancing. Prior work addresses this trade-off through request scheduling. However, scheduling cannot eliminate the fundamental conflict because hot KV cache couples reuse benefits with instance load, preventing requests from simultaneously achieving low computation time through reuse and low queueing delay through load balancing. Our key insight is to manage KV cache directly by replicating hot KV blocks across instances, preventing hot cache from causing load imbalance. We first define KV cache access pressure to characterize how cache distribution affects future request load across instances and show that balancing access pressure decouples KV cache reuse benefits from instance load. Based on this analysis, we propose KVFlow, the first KV cache management system that balances access pressure across instances. Across multiple LLMs and real LLM serving workloads, KVFlow reduces average time-to-first-token (TTFT) by 12.5%–54.3% compared with state-of-the-art production systems, including vLLM, Dynamo, and LMetric.

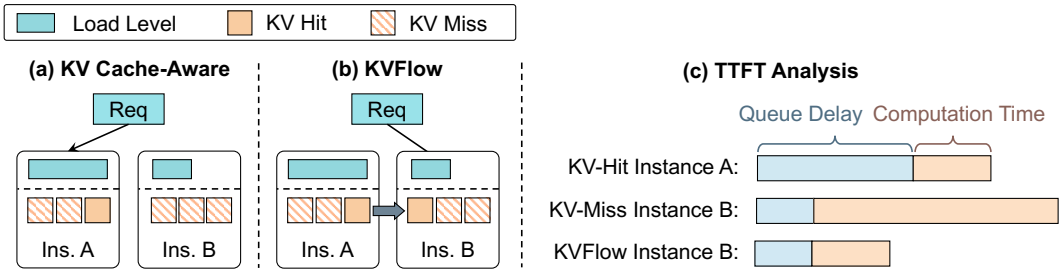

Figure 1: Examples of KV cache-aware scheduling and KVFlow. KVFlow enables requests to achieve both low queueing delay and low computation time.

1 INTRODUCTION

We study multi-instance large language model (LLM) serving. As LLMs are increasingly used in online applications, including chat assistants (Choi et al., 2024; Khurana et al., 2024), code generation (Wang et al., 2021; Li et al., 2022), and agents (Yao et al., 2023; Qin et al., 2024), serving systems must handle many concurrent requests. In practice, an instance is the basic execution unit for inference and typically holds a full copy of the model parameters. A cluster runs multiple instances, with a global scheduler dispatching incoming requests among them (Zhang et al., 2026; Qin et al., 2025; Xiang et al., 2025; NVIDIA, 2026; vLLM Project, 2026; Xiang et al., 2026). Each request consists of prefill, which processes the input and generates the first token, and decode, which generates subsequent tokens autoregressively. Serving latency is commonly measured by time-to-first-token (TTFT) and time-per-output-token (TPOT), which capture first-token and subsequent-token latency, respectively.

Multi-instance LLM serving must balance KV cache reuse and load. When a request hits reusable KV states on an instance, the matched tokens need not be recomputed, reducing prefill computation. However, routing too many hot requests to a few instances increases their load and queue-

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.

ing delay. Existing systems, including Dynamo (NVIDIA, 2026), LMetric (Zhang et al., 2026), SGLang (Zheng et al., 2024), and DualMap (Yuan et al., 2026), therefore route requests toward instances with reusable KV cache while limiting request concentration on hot instances, trading off KV cache reuse and load balancing.

However, even an ideal scheduler faces a fundamental bottleneck: it can only trade off KV cache reuse against load balancing, rather than simultaneously achieving the low computation time from KV cache reuse and the low queueing delay from balanced load (§ 3). Consider two instances, A and B, where A holds the KV cache of a hot prefix and B does not. Assume the scheduler always routes each request to the instance with the lowest latency. It initially prefers A because KV cache reuse reduces computation time, but as requests accumulate, A's queueing delay increases. Once this delay exceeds the computation benefit of KV cache reuse, the scheduler routes new requests to B. As shown in Figure 1(a) and (c), most requests must therefore choose between shorter computation time and shorter queueing delay, rather than obtaining both the low computation time of A and the low queueing delay of B.

We argue that balancing KV cache reuse and load requires shifting the focus from request scheduling to KV cache management (§3). The fundamental limitation of scheduling is that the reuse benefit of hot KV cache is coupled with instance load. Our key insight is to break this coupling by replicating hot KV cache across more instances, preventing it from becoming a source of load imbalance. As shown in Figure 1(b) and (c), replicating the reusable KV cache from instance A to B allows the request to be routed to B with both low queueing delay and low computation time.

To study how to distribute hot KV cache, we first define KV cache access pressure to characterize how cache distribution affects future request load across instances. We show that balancing access pressure not only decouples KV cache reuse from instance load, but also turns reactive, miss-driven cache expansion under request scheduling into proactive redistribution, avoiding repeated queue buildup and recomputation (§4).

To balance KV cache access pressure across instances during serving, we propose KVFlow, a KV cache management system for multi-instance LLM serving (§ 5). KVFlow first maintains a global KV cache index that compactly represents cache metadata while supporting heat tracking, access-pressure estimation, and cache management decisions. Based on this index, KVFlow continuously computes the KV cache access pressure of each instance. When the gap between the highest- and lowest-pressure instances exceeds a threshold, KVFlow selects hot KV cache blocks with high replication benefit from the former and replicates them to the latter. The threshold ensures that pressure differences reflect stable reuse patterns rather than short-term access fluctuations, improving the stability and effectiveness of replication decisions.

Balancing access pressure requires KV cache replication and eviction, both performed in the background without blocking ongoing requests. Since replication benefits future requests, it is not latency-sensitive and does not require high-bandwidth networks, allowing KVFlow to operate under diverse network conditions. Eviction targets KV cache blocks least likely to be reused, with negligible impact on the cache hit rate.

We implement KVFlow as a KV cache management layer beneath the inference engine. KVFlow is compatible with different KV cache-aware request schedulers, including Dynamo (NVIDIA, 2026) and LMetric (Zhang et al., 2026). By balancing KV cache access pressure, KVFlow allows existing schedulers to more often route requests to instances with both low computation time and low queueing delay.

We evaluate KVFlow with multiple models, including Qwen3-30B-A3B (Qwen Team, 2025) and DeepSeek-V2-Lite (DeepSeek-AI, 2024), and real LLM serving workloads from code agents (Jimenez et al., 2024) and general agents (Mialon et al., 2024). Compared with state-of-the-art production systems, including vLLM (Kwon et al., 2023), Dynamo, and LMetric, KVFlow reduces average TTFT by 12.5%–54.3% (§7).

In summary, our contributions are as follows:

- Identifying that balancing KV cache reuse and load requires KV cache management rather than request scheduling alone.

- Defining KV cache access pressure and proposing KVFlow, the first KV cache management system that balances access pressure across instances for efficient LLM serving.
- Validating KVFlow through extensive experiments and analysis, showing that it reduces average TTFT by 12.5%–54.3% over state-of-the-art systems.

2 Background and Related Work

LLM inference and KV cache. LLM inference has two stages: prefill processes the input and produces the first token, while decode generates subsequent tokens autoregressively. Service quality is measured by TTFT and TPOT, which capture first-token and per-token latency. After prefill, the key and value states of each token in each attention layer are stored in the KV cache. During decode, generating a new token x_t reuses the cached states of previous tokens $(K_{<t}, V_{<t})$, avoiding redundant computation.

The basic unit of LLM serving is an inference instance, which holds a complete copy of the model parameters. An instance retains KV caches from previous requests for reuse by subsequent requests, reducing TTFT. As the service continuously receives new requests, the amount of cached KV data grows over time. Many KV cache management systems therefore use both GPU high-bandwidth memory (HBM) and larger-capacity host memory to store KV caches (Cheng et al., 2025; Lee et al., 2024; Jiang et al., 2025; Qin et al., 2025; Gao et al., 2024).

LLM serving in a cluster. To handle large request volumes, LLM service providers typically deploy multiple inference instances in a cluster (Sun et al., 2024; Srivatsa et al., 2025; Zhang et al., 2026; Yuan et al., 2026). A global scheduler routes each request to an instance for execution. We focus on the common setting where both prefill and decode of a request run on the same instance. Although some systems disaggregate prefill and decode across different instances (Patel et al., 2024; Zhong et al., 2024), this design requires high network bandwidth. In production LLM serving, processing both stages on the same instance remains widely used (Xiang et al., 2025; Zhang et al., 2026).

Request scheduling policy. Scheduling requests across LLM instances requires balancing KV cache reuse and load. Traditional schedulers (Patel et al., 2013; Eisenbud et al., 2016; Olteanu et al., 2018; vLLM Project, 2026) that consider only load can cause frequent KV cache misses, increasing computation time. Conversely, prioritizing only KV cache hits can overload popular instances and increase queueing delay. Therefore, existing LLM schedulers, such as Dynamo (NVIDIA, 2026), LMetric (Zhang et al., 2026), SGLang (Zheng et al., 2024), and DualMap (Yuan et al., 2026), balance KV cache reuse and load. Their scheduling policies aim to route each request to the instance with the lowest combined computation time and queueing delay.

3 Motivation

Limitations of request scheduling. Even an ideal scheduler faces an inherent trade-off between queueing delay and computation time. Let q_i and c_i denote the queueing delay and computation time of instance i, respectively, with TTFT $l_i = q_i + c_i$. Suppose A holds reusable KV cache for popular requests while B does not, so $c_A < c_B$. Initially, $q_A \approx q_B$, yielding $l_A < l_B$, and requests are routed to A. As they accumulate, q_A increases until $q_A - q_B > c_B - c_A$, making $l_A > l_B$ and shifting subsequent requests to B. Thus, A minimizes computation time before the switch, while B minimizes queueing delay after it. In general, $\arg\min_i c_i \neq \arg\min_i q_i$, so the scheduler cannot minimize both simultaneously, as shown in Figure 1(a) and (c).

Opportunities in KV cache management. Our analysis shows that this trade-off arises from the coupling between KV cache reuse and instance load: an instance caching popular KV entries attracts more requests, increasing its load.

We therefore shift the focus from request scheduling to KV cache management. Our key insight is to distribute KV caches across instances so that cache reuse no longer causes load imbalance. Let c_i' denote the computation time after KV cache distribution. As shown in Figure 1(b) and (c), replicating reusable KV cache from A to B reduces B's computation time from c_B to $c_B' \approx c_A$, giving both instances similar reuse benefits. The scheduler can then select the less-loaded instance

based only on queueing delay. When $q_B < q_A$, routing the request to B achieves both $q_B = \min_i q_i$ and $c'_B \approx \min_i c_i$.

4 KV CACHE DISTRIBUTION ANALYSIS

This section studies how to distribute KV caches to decouple reuse benefits from instance load. We first define KV cache access pressure to capture the impact of cache distribution on future load, then show that balancing access pressure achieves this decoupling, and finally analyze practical pressure balancing to guide the system design.

4.1 FORMULATION OF KV CACHE ACCESS PRESSURE

KV cache reuse operates at the KV block level, the basic management unit in modern inference engines such as vLLM (Kwon et al., 2023). Each block stores the KV cache of multiple tokens, such as 256, and reuse matches the request prefix against existing blocks.

We define the access heat of KV block k as H_k, the number of cache hits to the block. Let r_k denote the number of replicas of block k, i.e., the number of instances storing it. Assuming accesses are evenly distributed across replicas, average access pressure on each replica is $p_k = H_k/r_k$. Let R_k denote the set of instances storing block k. We define KV cache access pressure of instance i as

$$P_i = \sum_{k:i\in R_k} p_k \tag{1}$$

Thus, P_i aggregates the access pressure of all KV block replicas stored on instance i and estimates its affinity to future requests. A larger P_i indicates that instance i stores more frequently accessed KV blocks and is therefore more likely to attract future requests through KV cache hits.

4.2 DECOUPLING KV CACHE REUSE BENEFIT FROM INSTANCE LOAD

When access pressure is balanced, i.e., $P_i = P_j$ for all instances, KV cache reuse benefit and instance load are decoupled. Balanced access pressure gives each instance the same expected cache affinity and thus the same expected KV cache reuse benefit. With homogeneous instances and equal-sized KV blocks, their expected computation times are equal:

$$\mathbb{E}[c_i] = \mathbb{E}[c_j], \qquad \forall i, j. \tag{2}$$

For a cache-aware scheduler that considers both cache affinity and load, the cache term is therefore identical in expectation, so scheduling is determined by load. If the load term is monotonic in queueing delay, the selected instance i^* satisfies

$$i^* \in \arg\min_i \mathbb{E}[q_i] \quad \text{and} \quad i^* \in \arg\min_i \mathbb{E}[c_i]. \tag{3}$$

Therefore, balancing access pressure removes the systematic trade-off between KV cache reuse and load balancing. More fundamentally, proactive replication turns KV cache placement from reactive, miss-driven expansion into proactive redistribution, avoiding the queue buildup and recomputation incurred by request scheduling alone. Detailed benefit analysis is provided in §A.1.

4.3 BALANCING ACCESS PRESSURE ACROSS INSTANCES

We analyze how to balance access pressure in practical serving to guide our design. Rebalancing is triggered only when the pressure gap between instances exceeds a threshold, since access pressure is estimated from historical reuse. Sufficient observations are needed to reliably reflect future reuse patterns and avoid rebalancing caused by random fluctuations.

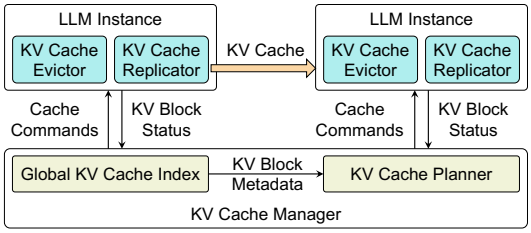

Figure 2: KVFlow overview.

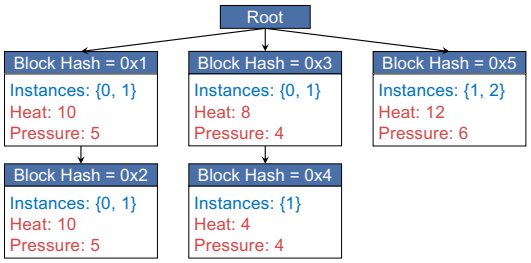

Figure 3: Example of the global KV cache index.

5 KVFLOW

5.1 OVERVIEW

Figure 2 shows the overview of KVFlow. The KV cache manager coordinates KV caches across instances through a global index that tracks their metadata. Based on this metadata, the KV cache planner generates replication and eviction plans to balance access pressure. The manager then sends the plans to the relevant instances, where local replicators and evictors execute them.

5.2 GLOBAL KV CACHE INDEX

As shown in Figure 3, the global KV cache index is a radix tree that tracks KV cache metadata across instances. The root represents an empty prefix, while each non-root node represents a cached KV block identified by its block hash, consistent with modern inference engines such as vLLM. A root-to-node path represents the KV block sequence of a request prefix, capturing both block metadata and prefix dependencies. Each node records the instances holding the block, its access heat H_k, and per-replica access pressure p_k, as defined in §4.1.

The radix tree captures prefix reuse, where a KV block is reusable only if its preceding blocks are available. By merging shared prefixes, it avoids duplicate metadata while preserving block dependencies for cache management. Replication can prioritize continuous prefixes and avoid blocks with absent dependencies, while eviction can favor tail blocks with less impact on prefix reuse. The index compactly supports heat tracking, access pressure computation, replication, and eviction.

5.3 KV CACHE PLANNER

The KV cache planner generates adjustment plans from the global KV cache index. When a block's heat changes, it recomputes each instance's access pressure. Based on the analysis in §4.3, if the gap between the highest- and lowest-pressure instances exceeds a threshold, the planner selects blocks to replicate from the former to the latter. If the target lacks sufficient cache space, it also selects low-value blocks for eviction with minimal impact on future prefix reuse.

Adjustment trigger condition. The planner triggers KV cache adjustment when the pressure gap between the highest- and lowest-pressure instances exceeds βC_{gpu}, where C_{gpu} is the maximum number of KV blocks that an instance can store in GPU memory and β is the imbalance tolerance factor. We scale the threshold by C_{gpu} because cache capacity determines the space available for online request execution and thus reflects the imbalance an instance can tolerate.

We identify two properties of β. First, performance is insensitive to small changes in β, so a near-optimal value requires only a few coarse adjustments. Second, β should increase with the prefix reuse rate. Detailed analysis is provided in §A.2, and both properties are validated in §7.3.

KV block selection for replication. After adjustment is triggered, the planner selects candidate KV blocks for replication. A candidate must exist on the source but not the target, and its preceding blocks must either exist on the target or be included in the candidate set. This ensures that replication forms reusable continuous prefixes.

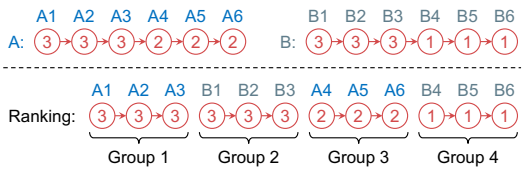

Figure 4: Example of KV block sorting by replication benefit Δp_k, with consecutive blocks from the same request prefix grouped together.

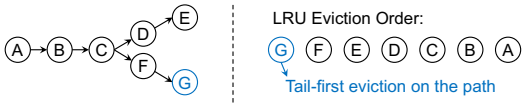

Figure 5: Example of KV block eviction candidates.

The planner ranks candidates by the reduction in access pressure from adding one replica:

$$\Delta p_k = \frac{H_k}{r_k} - \frac{H_k}{r_k + 1}. \tag{4}$$

Here, H_k and r_k denote the access heat and current replica count of block k, respectively. A larger Δp_k indicates a greater pressure reduction and thus higher replication priority. For candidates with the same Δp_k, consecutive blocks from the same request prefix are grouped to preserve prefix continuity. For example, in Figure 4, A1–A3 are ranked as a group before B1–B3.

The planner selects groups to reduce the pressure gap while accounting for the pressure change caused by blocks evicted from the target. It does not require exact balance because KV blocks are discrete and the last selected group is kept intact to preserve reuse opportunities. For the example in Figure 4, even if replicating only A4 or A4–A5 yields a smaller pressure gap, the planner replicates A4–A6 because these blocks have the same replication priority and reuse opportunity.

Replication gains diminish rapidly, so most KV blocks have fewer than three replicas, limiting cache space consumed by over-replication. We analyze this property in §A.3 and validate it in §7.4.

KV block selection for eviction. When the target lacks space for new replicas, the planner generates an eviction plan using LRU and evicts enough blocks to accommodate them. It also preserves prefix dependencies: because evicting an earlier block makes subsequent blocks on the same prefix path unusable, the planner prefers tail blocks when candidates have the same LRU priority. In Figure 5, if F and G have the same priority and only one block must be evicted, the planner evicts G.

Eviction also updates access heat by removing the pressure assigned to the evicted replica. Specifically, evicting one replica updates the block heat as $H'_k = H_k - p_k = H_k - H_k/r_k$. This makes heat time-aware: stale blocks gradually become LRU candidates, and once evicted, their assigned pressure is removed from the index, preventing outdated reuse history from biasing subsequent pressure estimates and cache adjustments.

Eviction has negligible impact on the cache hit rate, as analyzed in §A.4 and validated in §7.4.

6 Implementation

We implement a KVFlow prototype on vLLM (Kwon et al., 2023) and LMCache (Cheng et al., 2025). KVFlow has two components: a standalone KV cache manager that maintains global metadata and generates replication and eviction plans, and per-instance KV cache replicators and evictors integrated into LMCache-enabled vLLM instances to execute these plans. Both are implemented by extending LMCache, decoupling global cache management from local inference and avoiding intrusive changes to the vLLM execution pipeline.

The manager incrementally maintains the global KV cache index by subscribing to KV cache events from vLLM instances. Each event reports metadata such as the block hash, parent hash, token information, block size, and storage medium, allowing the manager to track block identity, prefix dependencies, and location without blocking online inference.

KVFlow mainly manages CPU-resident KV cache. GPU memory is typically occupied by model parameters, activations, and KV cache of running requests, leaving limited space for long-lived reusable cache, while LMCache uses larger CPU memory for cross-request reuse. KVFlow therefore balances access pressure by replicating and evicting CPU-resident KV blocks across instances.

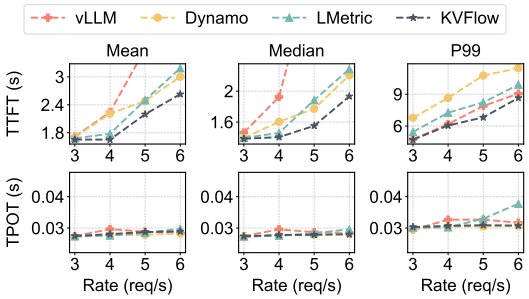

Figure 6: TTFT and TPOT comparison for Qwen3-30B-A3B on SWE-bench under different request rates.

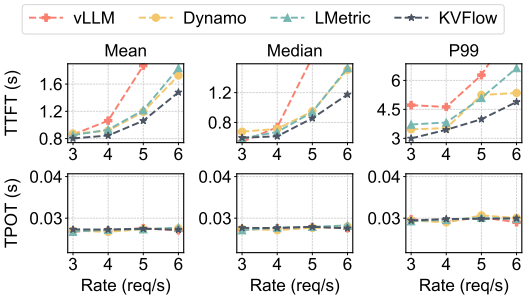

Figure 7: TTFT and TPOT comparison for Qwen3-30B-A3B on GAIA under different request rates.

7 Evaluation

This section evaluates KVFlow against state-of-the-art baselines using representative models and datasets. We further study its effectiveness through ablation, overhead analysis, and network sensitivity experiments.

7.1 Experimental Setup

Testbed. We deploy eight instances on eight machines, each with an NVIDIA RTX PRO 6000 GPU and 80 GB of CPU memory. The machines are connected by a 10 Gbps network. We further evaluate network sensitivity at 1 Gbps and 500 Mbps.

Models. We evaluate KVFlow on Qwen3-30B-A3B (Qwen Team, 2025) and DeepSeek-V2-Lite (DeepSeek-AI, 2024).

Traces and datasets. We use real agentic traces from the LMCache Agentic Dataset (LMCache, 2026), including SWE-bench (Jimenez et al., 2024) and GAIA (Mialon et al., 2024) workloads. Their prefix reuse rates are about 94% and 72%, respectively. Since the dataset lacks request arrival times, we adopt real arrival times from the Azure Conversation Dataset (Patel et al., 2024) and scale the inter-arrival times to generate different average request rates.

Baselines. We compare KVFlow with three competitive baselines. All systems use the same vLLM/LMCache cache backend.

1) vLLM (Kwon et al., 2023): A production-grade LLM serving system that routes requests to the least-loaded instance.

2) Dynamo (NVIDIA, 2026): A production-grade KV cache-aware scheduler that jointly considers cache reuse and instance load. We tune its weighting parameter between cache hits and load to the optimal setting.

3) LMetric (Zhang et al., 2026): A state-of-the-art LLM scheduler that routes each request to the instance minimizing the product of KV-missed tokens and unfinished requests, avoiding the tuning required by weighted trade-off schemes.

In the performance comparison, KVFlow uses Dynamo as the scheduler. In the ablation study, we also evaluate its gains when using LMetric as the scheduler.

Metrics. We report TTFT and TPOT for overall performance, and KV cache hit rate and load imbalance to analyze performance gains.

7.2 Overall Performance

Figure 6 shows the mean, median, and P99 TTFT and TPOT of Qwen3-30B-A3B on SWE-bench. KVFlow consistently reduces TTFT over vLLM, Dynamo, and LMetric, with larger gains at higher request rates, where longer queues amplify the trade-off between KV cache reuse and load balancing. vLLM favors load balance but sacrifices reuse, while Dynamo and LMetric exploit reuse but can

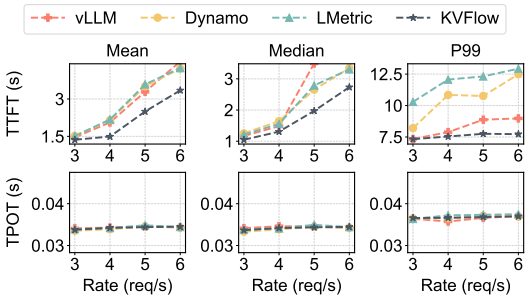

Figure 8: TTFT and TPOT comparison for DeepSeek-V2-Lite on SWE-bench under different request rates.

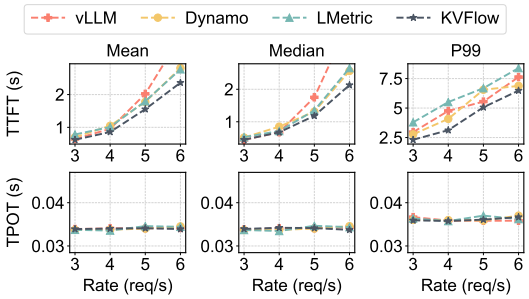

Figure 9: TTFT and TPOT comparison for DeepSeek-V2-Lite on GAIA under different request rates.

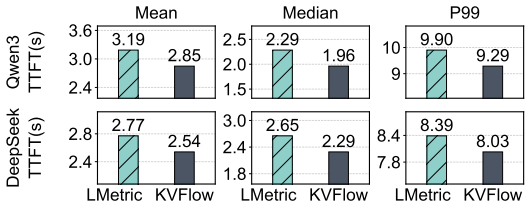

Figure 10: TTFT comparison with LMetric at 6 req/s. The top row uses Qwen3-30B-A3B on SWE-bench, and the bottom row uses DeepSeek-V2-Lite on GAIA.

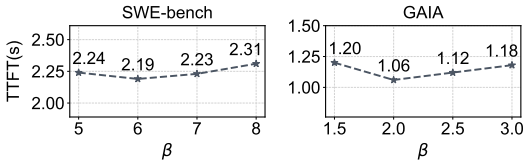

Figure 11: Parameter ablation of the imbalance tolerance factor β on Qwen3-30B-A3B.

concentrate hot-prefix requests on a few instances. By balancing KV cache access pressure, KVFlow preserves high cache hit rates while reducing hotspot queueing delay. At 6 req/s, it reduces average TTFT by 45%, 12.5%, and 17.5% over vLLM, Dynamo, and LMetric, respectively. TPOT remains similar across systems because access pressure balancing mainly improves prefill efficiency and has limited impact on decode.

Figure 7, Figure 8, and Figure 9 show consistent results for Qwen3-30B-A3B on GAIA and DeepSeek-V2-Lite on SWE-bench and GAIA. At 6 req/s, KVFlow reduces average TTFT over vLLM, Dynamo, and LMetric by 54.3%, 14.3%, and 19.4% for Qwen3-30B-A3B on GAIA, by 25.8%, 20.7%, and 21.1% for DeepSeek-V2-Lite on SWE-bench, and by 37.1%, 15.9%, and 14.8% for DeepSeek-V2-Lite on GAIA, respectively.

7.3 Ablation Studies

Scheduler-agnostic benefits. We further integrate KVFlow with LMetric and evaluate it at 6 req/s to verify that its benefits are not tied to a specific scheduler. As shown in Figure 10, KVFlow reduces average TTFT by 10.7% for Qwen3-30B-A3B on SWE-bench and 8.3% for DeepSeek-V2-Lite on GAIA compared with LMetric. These results confirm that KVFlow provides consistent benefits across different KV cache-aware scheduling policies.

Parameter ablation. Figure 11 evaluates Qwen3-30B-A3B under different imbalance tolerance factors β and validates the properties identified in § 5.3. First, performance is insensitive to β, allowing a near-optimal value to be found with a few coarse trials. On SWE-bench, average TTFT is 2.24 s, 2.19 s, and 2.23 s for $\beta = 5$, 6, and 7, respectively, showing limited benefit from finer tuning. Second, workloads with higher prefix reuse rates favor larger β values. The optimal β is 6 on SWE-bench and 2 on GAIA.

7.4 Overhead Analysis

KV cache hit rate analysis. Figure 12 compares the KV cache hit rates of Dynamo and KVFlow on Qwen3-30B-A3B with GAIA at 5 req/s. Both achieve comparable hit rates, supporting the argument in §5.3 that eviction has negligible impact on the cache hit rate.

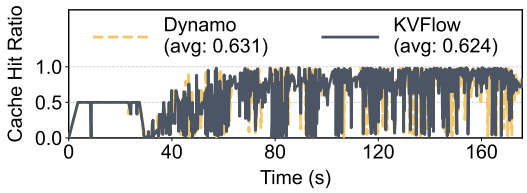

Figure 12: KV cache hit ratio over time for Qwen3-30B-A3B on GAIA at 5 req/s.

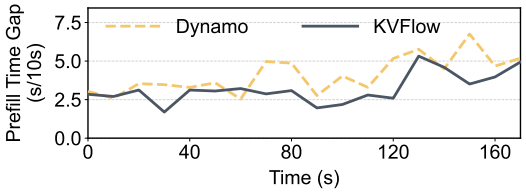

Figure 13: Prefill load imbalance over time for Qwen3-30B-A3B on GAIA at 5 req/s. The y-axis reports the per-10s gap between the most and least loaded prefill instances.

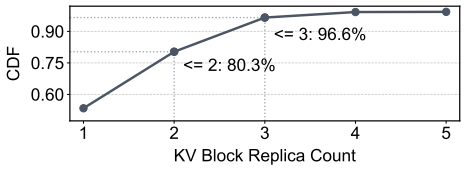

Figure 14: CDF of KV block replica counts for Qwen3-30B-A3B on SWE-bench at 6 req/s.

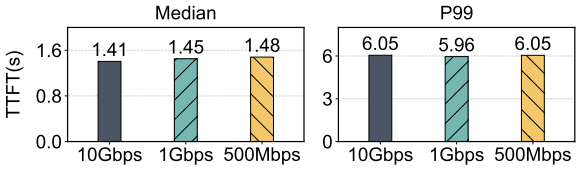

Figure 15: Median and P99 TTFT of Qwen3-30B-A3B on SWE-bench at 4 req/s under 10 Gbps, 1 Gbps, and 500 Mbps links.

The gains of KVFlow therefore mainly come from mitigating cache-induced load concentration and reducing queueing delay. We measure load imbalance as the gap between the maximum and minimum prefill time across instances within each 10 s window, where a larger gap indicates greater imbalance. As shown in Figure 13, KVFlow achieves better load balance than Dynamo in most cases, confirming that it reduces queueing delay while maintaining a high KV cache hit rate.

KV block replication analysis. As noted in §5.3, KV blocks tend to maintain few replicas. Figure 14 shows the CDF of KV block replica counts for Qwen3-30B-A3B on SWE-bench. Overall, 80.3% of blocks have at most two replicas and 96.6% have at most three, showing that KVFlow disperses the access pressure of hot blocks without excessive cache space overhead.

Each KV block adjustment takes 2.98 s on average and copies 102 blocks, each containing 256 tokens. Adjustments run in the background without blocking inference, while the planner takes only 0.86 ms on average to generate each plan. Thus, planning overhead is negligible and does not bottleneck KV block adjustment.

7.5 Sensitivity Analysis

We evaluate network bandwidth sensitivity in Figure 15. Performance remains similar under 10 Gbps, 1 Gbps, and 500 Mbps: lower bandwidth increases median TTFT by only a few tens of milliseconds and does not increase P99 TTFT. This is because bandwidth affects only KV block replication time, delaying when access pressure balancing takes effect rather than reducing its benefits. During this delay, only a small number of hot-prefix requests miss these benefits, with limited impact on the overall TTFT distribution. These results indicate that KVFlow does not rely on high-speed networks and can adapt to a wide range of multi-instance serving environments.

8 Conclusion

We present KVFlow, a KV cache management system for multi-instance LLM serving. KVFlow balances KV cache access pressure by replicating hot KV blocks across instances, breaking the inherent trade-off between cache reuse and load balancing in request scheduling. By distributing hot KV cache across more instances, KVFlow enables high cache hit rates and low queueing delay, reducing TTFT and improving serving performance. Our results highlight KV cache management as a complementary optimization dimension beyond request scheduling.

AI USE STATEMENT

Generative AI tools were used solely to assist with translating portions of the manuscript into English and polishing the writing for grammar, clarity, and readability. They were not used for research ideation, methodology or experiment design, method implementation, data generation or analysis, or interpretation of results. All AI-assisted translations and edits were carefully reviewed and revised by the authors for accuracy and fidelity to the original content. The authors take full responsibility for the final content of this work.

REFERENCES

Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. LMCache: An efficient KV cache layer for enterprise-scale LLM inference, 2025. https://arxiv.org/abs/2510.09665.

Dasom Choi, Sunok Lee, Sung In Kim, Kyungah Lee, Hee Jeong Yoo, Sangsu Lee, and Hwajung Hong. Unlock life with a Chat(GPT): Integrating conversational AI with large language models into everyday lives of autistic individuals. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pp. 1–17. ACM, 2024. doi: 10.1145/3613904.3641989. https://dl.acm.org/doi/10.1145/3613904.3641989.

DeepSeek-AI. DeepSeek-V2-Lite, 2024. https://huggingface.co/deepseek-ai/DeepSeek-V2-Lite.

Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 523–535, Santa Clara, CA, 2016. USENIX Association. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 111–126, Santa Clara, CA, 2024. USENIX Association. ISBN 978-1-939133-41-0. https://www.usenix.org/conference/atc24/presentation/gao-bin-cost.

Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. NEO: Saving GPU memory crisis with CPU offloading for online LLM inference. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. https://proceedings.mlsys.org/paper_files/paper/2025/hash/66a026c0d17040889b50f0dfa650e5e0-Abstract-Conference.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *Proceedings of the International Conference on Learning Representations (ICLR 24)*, 2024. https://openreview.net/forum?id=VTF8yNQM66.

Anjali Khurana, Hari Subramonyam, and Parmit K. Chilana. Why and when LLM-Based assistants can go wrong: Investigating the effectiveness of prompt-based interactions for software help-seeking. In *Proceedings of the 29th International Conference on Intelligent User Interfaces*. ACM, 2024. doi: 10.1145/3640543.3645200. https://dl.acm.org/doi/10.1145/3640543.3645200.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. https://dl.acm.org/doi/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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 155–172, Santa Clara, CA, 2024. USENIX Association. ISBN 978-1-939133-40-3. https://www.usenix.org/conference/osdi24/presentation/lee.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. https://www.science.org/doi/10.1126/science.abq1158.

LMCache. LMCache Agentic Dataset: Multi-turn LLM agent sessions for KV cache benchmarking, 2026. https://huggingface.co/datasets/sammshen/lmcache-agentic-traces.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *Proceedings of the International Conference on Learning Representations (ICLR 24)*, 2024. https://openreview.net/forum?id=fibxvahvs3.

NVIDIA. NVIDIA Dynamo documentation, 2026. https://docs.nvidia.com/dynamo/.

Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with Beamer. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 125–139, Renton, WA, 2018. USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/olteanu.

Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pp. 207–218. ACM, 2013. doi: 10.1145/2534169.2486026. https://dl.acm.org/doi/10.1145/2534169.2486026.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Iñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA 2024)*. ACM, 2024. https://www.microsoft.com/en-us/research/wp-content/uploads/2023/12/Splitwise_ISCA24.pdf.

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, 2025. USENIX Association. ISBN 978-1-939133-45-8. https://www.usenix.org/conference/fast25/presentation/qin.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *Proceedings of the 12th International Conference on Learning Representations (ICLR 2024)*, 2024. https://openreview.net/forum?id=dHng2O0Jjr.

Qwen Team. Qwen3-30B-A3B-Instruct-2507, 2025. https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507.

John F. Shortle, James M. Thompson, Donald Gross, and Carl M. Harris. *Fundamentals of Queueing Theory*, volume 399 of *Wiley Series in Probability and Statistics*. John Wiley & Sons, 5 edition, 2018. doi: 10.1002/9781119453765. https://onlinelibrary.wiley.com/doi/book/10.1002/9781119453765.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. In *Proceedings of the 13th International Conference on Learning Representations (ICLR 2025)*, 2025. https://openreview.net/forum?id=meKEKDhdnx.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 173–191. USENIX Association, 2024. https://www.usenix.org/conference/osdi24/presentation/sun-biao.

vLLM Project. Data Parallel Deployment, 2026. https://docs.vllm.ai/en/latest/serving/data_parallel_deployment/.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. https://aclanthology.org/2021.emnlp-main.685/.

Yuxing Xiang, Zhe Li, Ji Qi, Hongrong Yu, Jiacheng Zhao, Fan Yang, Jingren Zhou, Qiang Li, Ennan Zhai, and Enhong Chen. Aegaeon: Effective GPU pooling for concurrent LLM serving on the market. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 25)*, pp. 1030–1045. ACM, 2025. doi: 10.1145/3731569.3764815. https://dl.acm.org/doi/10.1145/3731569.3764815.

Yuxing Xiang, Xue Li, Kun Qian, Yan Zhang, Wenyuan Yu, Ennan Zhai, Xin Jin, and Jingren Zhou. ServeGen: Workload characterization and generation of large language model serving in production. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 1845–1859, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. https://www.usenix.org/conference/nsdi26/presentation/xiang-servegen.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR 2023)*, 2023. https://openreview.net/forum?id=WE_vluYUL-X.

Ying Yuan, Pengfei Zuo, Bo Wang, Zhangyu Chen, Zhipeng Tan, and Zhou Yu. DualMap: Enabling both cache affinity and load balancing for distributed LLM serving. In *Proceedings of the 14th International Conference on Learning Representations (ICLR 2026)*, 2026. https://openreview.net/forum?id=zCadrJ32Xn.

Dingyan Zhang, Jinbo Han, Kaixi Zhang, Xingda Wei, Sijie Shen, Chenguang Fang, Wenyuan Yu, Jingren Zhou, and Rong Chen. Simple is better: Multiplication may be all you need for LLM request scheduling. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, 2026. https://www.usenix.org/conference/osdi26/presentation/zhang-dingyan.

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 *Advances in Neural Information Processing Systems*, 2024. https://proceedings.neurips.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 *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, Santa Clara, CA, 2024. USENIX Association. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

A Additional Analysis

A.1 Benefit Analysis of Access Pressure Balancing

We first analyze the average benefit of access pressure balancing from a static perspective, assuming KV cache is not redistributed across instances through recomputation. We then take a dynamic perspective to identify which requests realize this benefit during actual inference.

Static analysis. We build a simplified cache-load coupling model. Assume N instances and a class of requests associated with hot KV cache arriving at an average rate λ following a Poisson process. For clarity, we first consider only this request class. The prefill latency is C_{miss} on a KV cache miss and is approximated as $(1-h)C_{\text{miss}}$ on a hit, where h is the KV cache hit rate.

Under load-based scheduling, requests are evenly distributed across the N instances, giving each instance an arrival rate of approximately λ/N. Since scheduling ignores KV cache reuse, the prefill latency is approximated as C_{miss}. With continuous batching, request queueing can be modeled as an M/D/1 queue. We assume each instance satisfies:

$$\rho_{\text{load}} = \frac{\lambda C_{\text{miss}}}{N} < 1 \tag{5}$$

According to the M/D/1 queueing model (Shortle et al., 2018), the average TTFT is:

$$\bar{T}_{\text{load}} = C_{\text{miss}} + \frac{\rho_{\text{load}} C_{\text{miss}}}{2(1-\rho_{\text{load}})} \tag{6}$$

For KV cache-aware scheduling, the scheduler preferentially routes hot requests to instances holding the corresponding KV cache. Assume the hot KV cache exists on only r_0 instances, where typically $r_0 \ll N$. Ideally, requests are evenly distributed across these instances, giving each an arrival rate of approximately λ/r_0. With KV cache hits, the prefill latency is approximated as $(1-h)C_{\text{miss}}$. We assume each cached instance satisfies:

$$\rho_{\text{kv}} = \frac{\lambda(1-h)C_{\text{miss}}}{r_0} < 1 \tag{7}$$

According to the M/D/1 queueing model, the average TTFT is:

$$\bar{T}_{\text{kv}} = (1-h)C_{\text{miss}} + \frac{\rho_{\text{kv}}(1-h)C_{\text{miss}}}{2(1-\rho_{\text{kv}})} \tag{8}$$

With an ideal scheduler, $\bar{T}_{\text{kv}}$ does not exceed $\bar{T}_{\text{load}}$. When cached instances become overloaded and their queueing delay grows too large, the scheduler falls back to load-based routing.

With KV cache access pressure balancing, this hot KV cache is distributed across all N instances because no other hot KV cache class exists. Requests are therefore distributed at rate λ/N per instance. We assume each instance satisfies:

$$\rho_{\text{bal}} = \frac{\lambda(1-h)C_{\text{miss}}}{N} < 1 \tag{9}$$

According to the M/D/1 queueing model, the average TTFT is:

$$\bar{T}_{\text{bal}} = (1-h)C_{\text{miss}} + \frac{\rho_{\text{bal}}(1-h)C_{\text{miss}}}{2(1-\rho_{\text{bal}})} \tag{10}$$

We use an example to illustrate the benefits of balancing access pressure. We set $N = 8$, $C_{\text{miss}} = 1$, and $h = 0.8$. For KV cache-aware scheduling, we assume the hot KV cache exists on one instance. Figure 16 shows the average TTFT trends in the three cases.

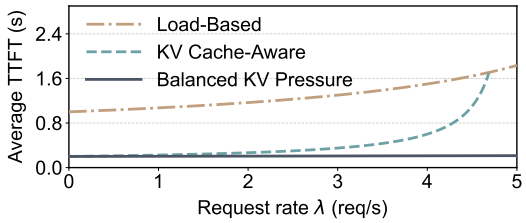

Figure 16: Theoretical TTFT of load-based scheduling, KV cache-aware scheduling, and KV cache access pressure balancing under different request rates.

Balancing access pressure effectively reduces the queueing overhead of KV cache-aware scheduling. As the request rate increases, KV cache-aware scheduling quickly degrades toward load-based scheduling because of growing queueing delay. In contrast, access pressure balancing distributes hot KV cache across more instances, preserving the low computation time from KV cache hits while avoiding queueing delay caused by load concentration.

Dynamic analysis. In practice, KV cache placement evolves with request execution. Let $R_k(t)$ denote the set of instances storing the KV cache of prefix k at time t, with $r_k(t) = |R_k(t)|$. For request j of prefix k arriving at t_j and routed to instance i_j, define the recomputation indicator as $m_j = \mathbf{1}[i_j \notin R_k(t_j)]$. If $m_j = 1$, prefill generates the missing KV cache and, assuming no eviction, expands the replica set to $R_k(t_j^+) = R_k(t_j^-) \cup i_j$. Thus, without explicit redistribution, KV cache replicas expand only through cache-miss prefill.

Under load-based scheduling, routing is independent of $R_k(t)$. Starting from $r_k(0) = r_0$ replicas, spreading prefix k to all N instances requires $N - r_0$ cache-miss prefills on previously uncached instances. Since a miss costs C_{miss} and a hit costs $(1-h)C_{\text{miss}}$, each recomputation incurs an additional cost of $\Delta C = hC_{\text{miss}}$. The cumulative recomputation overhead before full replication is

$$O_{\text{load}}^{\text{recomp}} = (N - r_0)\Delta C = (N - r_0)hC_{\text{miss}}. \tag{11}$$

This overhead grows with the number of instances and recurs across hot prefixes. Access pressure balancing avoids these request-triggered recomputations by proactively distributing hot KV cache.

For KV cache-aware scheduling, let $q_i(t)$ denote the queueing delay of instance i at time t. For prefix k, define the minimum queueing delays among cached and uncached instances as $q_k^{\text{hit}}(t) = \min_{i \in R_k(t)} q_i(t)$ and $q_k^{\text{miss}}(t) = \min_{i \notin R_k(t)} q_i(t)$, respectively. An ideal scheduler routes a request to an uncached instance only when

$$q_k^{\text{hit}}(t) - q_k^{\text{miss}}(t) > \Delta C, \tag{12}$$

because the queueing advantage then exceeds the computation benefit ΔC from KV cache reuse. Before this condition holds, requests remain concentrated on cached instances and increase their queueing delay. Once it holds, a request spills over to an uncached instance, incurs cache-miss prefill, and expands $R_k(t)$. The system therefore alternates between queue buildup and miss-driven replica expansion.

Access pressure balancing breaks this cycle by distributing hot KV cache before queueing forces spillover. It therefore benefits requests that would otherwise wait on overloaded cached instances or incur cache-miss prefill on uncached instances. Across many hot prefixes, avoiding these repeated queueing and recomputation costs preserves the benefit predicted by the static analysis.

A.2 Properties of the Imbalance Tolerance Factor

The imbalance tolerance factor β has two key properties.

First, performance is insensitive to small changes in β. Let $\mathcal{A}(\beta)$ denote the KV blocks selected for replication when adjustment is triggered under β. Once a stable hotspot forms, high-pressure blocks

dominate the replication benefit, so a small perturbation δ typically results in $\mathcal{A}(\beta + \delta) \approx \mathcal{A}(\beta)$. The resulting cache distributions and their benefits to future requests are therefore similar. Since cache adjustment affects only future requests, small differences in trigger time also have limited impact. Thus, $G(\beta + \delta) \approx G(\beta)$ for small δ, allowing a near-optimal β to be found with only a few coarse-grained trials.

Second, β should increase with the prefix reuse rate. Let η denote the prefix reuse rate and $\sigma_P(\eta)$ the natural variation in access pressure across instances. As η increases, each reuse typically covers more KV blocks and contributes more pressure, increasing $\sigma_P(\eta)$. If β is too small relative to this variation, transient fluctuations may trigger adjustment before stable hotspots emerge, making cache placement sensitive to short-term reuse patterns. Therefore, β should scale with the natural pressure variation, i.e., $\beta \propto \sigma_P(\eta)$, suggesting a larger β for workloads with higher prefix reuse.

A.3 Diminishing Returns of KV Cache Replication

The marginal pressure reduction from adding one replica is

$$\Delta p_k(r_k) = \frac{H_k}{r_k} - \frac{H_k}{r_k + 1} = \frac{H_k}{r_k(r_k + 1)}. \tag{13}$$

For $H_k > 0$ and $r_k \geq 1$,

$$\frac{\Delta p_k(r_k + 1)}{\Delta p_k(r_k)} = \frac{r_k}{r_k + 2} < 1, \tag{14}$$

so the marginal replication benefit strictly decreases with r_k. In particular, increasing the replica count from 1 to 2, 2 to 3, and 3 to 4 reduces access pressure by $H_k/2$, $H_k/6$, and $H_k/12$, respectively. Thus, the second and third added replicas provide only $1/3$ and $1/6$ of the benefit of the first. This rapid diminishing return discourages over-replication and limits cache-space consumption.

A.4 Impact of Eviction on KV Cache Hit Rate

Balancing access pressure replicates hot KV cache from high-pressure to low-pressure instances. If the target lacks free cache space, existing blocks must be evicted. We argue that this has little impact on the overall KV cache hit rate.

Balancing access pressure mainly advances cache replacement that would otherwise occur through recomputation. Consider a hot prefix k absent from target instance i. Without proactive replication, once the queueing advantage of i exceeds the reuse benefit on cached instances, a request is routed to i and generates k through cache-miss prefill. Let t_r denote the time when balancing replicates k to i, and $t_m > t_r$ the time when this cache would otherwise be generated through recomputation. Both consume the same cache space and may trigger eviction when space is insufficient. Thus, balancing mainly shifts replacement from t_m to t_r rather than introducing additional long-term cache occupancy.

Any additional hit-rate loss is therefore limited to $[t_r, t_m)$. Let $\pi_b(t_r, t_m)$ denote the probability that an evicted block b is reused during this interval. For eviction set E_i, the additional miss probability is bounded by

$$\Delta M_i \leq \sum_{b \in E_i} \pi_b(t_r, t_m). \tag{15}$$

Under LRU, blocks in E_i are the least recently accessed and typically have low short-term reuse probability, making this bound small. In contrast, replicated blocks have high access pressure and are more likely to be reused. Access pressure balancing therefore replaces low-value blocks earlier with high-value ones while largely preserving the eventual cache replacement, resulting in little hit-rate loss.