

CacheSched: Efficient LLM Request Scheduling for Hierarchical Prefix Caching

Rilin Huang*
School of Computer Science
Peking University
Beijing, China
h_rilin@stu.pku.edu.cn

Junyi Shu*†
Computer Science
University of California, Los Angeles
Los Angeles, California, USA
kyleshu@cs.ucla.edu

Yaofeng Tu
Nanjing Branch of ZTE Corporation
Nanjing, China
tu.yaofeng1@goldendb.com

Xin Jin
School of Computer Science
Peking University
Beijing, China
xinjinpku@pku.edu.cn

Abstract

Stateful Large Language Model (LLM) applications heavily rely on prefix caching to eliminate redundant prefill computations. To support massive concurrency, modern serving engines deploy hierarchical, multi-tier caches that offload less-active prefixes from limited GPU memory to spacious host DRAM. However, existing multi-worker request schedulers typically route traffic based solely on the longest prefix match, ignoring the highly heterogeneous data transfer costs between cache tiers. This oversight causes severe cluster-level load imbalance and triggers frequent GPU pipeline stalls as workers wait for slow host-to-device data transfers.

To resolve these inefficiencies, we propose CacheSched, a scheduling system tailored for multi-worker LLM serving with hierarchical prefix caching. CacheSched is driven by a lightweight analytic performance model that estimates batch execution and data transfer times. Building on this model, CacheSched introduces two key mechanisms: (1) a heterogeneity-adaptive routing algorithm that dispatches requests based on predicted prefill costs, successfully balancing cache tier locality with cluster-wide worker load, and (2) a dynamic batch-matching algorithm at the worker level, which restricts host-memory prefix loads to the predicted compute window of the active batch, reducing GPU pipeline stalls. Evaluations demonstrate that CacheSched increases overall serving throughput by up to 2.58× compared to state-of-the-art systems while satisfying strict Time-to-First-Token (TTFT) Service-Level Objectives (SLOs).

CCS Concepts: • **Software and its engineering** → **Memory management**; • **Computing methodologies** → *Distributed computing methodologies*.

*Equal contribution.
†Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857048

Keywords: Large Language Models, Prefix Cache, Request Scheduling

ACM Reference Format:
Rilin Huang, Junyi Shu, Yaofeng Tu, and Xin Jin. 2026. CacheSched: Efficient LLM Request Scheduling for Hierarchical Prefix Caching. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857048

1 Introduction

The rapid adoption of Large Language Models (LLMs) has made them foundational to a diverse range of applications, including chatbots [4, 7], code comprehension and generation [5, 6], research assistants [8, 14], and potentially many others [20, 44]. Serving an LLM inference request involves a compute-intensive two-stage process: a *prefill* phase to process the entire input context in a single forward pass, followed by an autoregressive *decoding* phase to generate the response one token at a time. To avoid recomputing all previous tokens at every decoding step, the underlying inference engine caches intermediate states (i.e., the KV cache) for each token, and appends new entries as each token is generated. This computational efficiency, however, comes at the cost of high memory consumption, as the KV cache grows rapidly with sequence length and the number of concurrent requests. This trade-off makes managing the KV cache a central challenge in LLM serving, and notable open-source inference engines like vLLM [38], SGLang [66], LightLLM [9], and TensorRT-LLM [13] have incorporated virtual memory and paging to address this issue.

While LLMs are powerful in responding to individual, standalone queries, their true utility increasingly emerges in stateful, iterative applications where maintaining context across interactions is essential. In many such real-world applications, such as multi-round conversations and agentic AI, inference requests often share an identical sequence of tokens at the beginning. The inherently stateless nature of LLMs creates a significant performance bottleneck, as the full context must be prefilled again for each new request. To

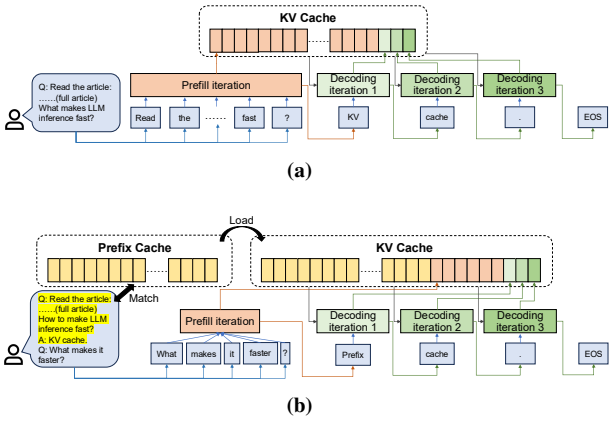


Figure 1. Accelerating LLM inference with prefix caching. (a) LLM inference with KV cache. (b) LLM inference with KV cache and prefix cache.

address this, prefix caching has become a critical optimization [12, 43, 52, 56, 66]. By storing and reusing the KV cache associated with the shared prefix, an LLM inference system avoids redundant calculations across requests and focuses only on processing the new content in each request, as illustrated in Figure 1. This efficiency makes prefix caching a cornerstone of stateful LLM serving. However, caching prefixes can easily exhaust GPU memory, thereby hindering the prefill and decoding phases of active requests by leaving insufficient space for their KV cache. To resolve this memory contention while simultaneously enabling a much larger total cache, inference systems implement a multi-tiered cache hierarchy that offloads less-active prefixes to host DRAM [24, 56, 59, 66], which typically provides at least an order of magnitude more capacity than GPU memory. The most active prefixes remain in GPU memory for fast access, while less-active ones are evicted to host DRAM and loaded back on demand.

In production, LLM serving scales to multiple GPU workers, each maintaining a hierarchical prefix cache spanning GPU memory and host DRAM. This multi-tier deployment introduces critical scheduling challenges at both the cluster and worker levels. At the *router level*, existing strategies, such as Preble [46] and SGLang Router [2], greedily rely on the longest prefix match, oblivious to the highly heterogeneous costs between instant GPU memory access and slow PCIe transfers from host DRAM. This ignorance, compounded by repetitive routing to the same worker in multi-turn sessions, frequently causes severe load imbalance. At the *worker level*, systems (e.g., SGLang) attempt to overlap host-to-GPU data transfers with computation using fixed prefetching strategies; however, because prefix sizes vary, long transfers inevitably outlast the compute phase and stall the GPU pipeline.

To overcome these inefficiencies, we propose CacheSched, a prefix-cache-aware request scheduling system that deeply integrates cluster-level routing and worker-level batch construction. CacheSched is built on a lightweight, unified analytic performance model. Our core insight is that while autoregressive decoding lengths are non-deterministic, the execution time of a specific GPU compute batch and the PCIe transfer time of a given prefix size are highly predictable. By profiling effective GPU memory bandwidth and effective PCIe transfer rates—rather than relying on idealized nominal hardware specifications—our model estimates both the compute boundaries and data transfer costs.

Building on this model, CacheSched implements two highly coordinated mechanisms:

First, a *heterogeneity-adaptive request routing algorithm* shifts the routing paradigm from maximizing prefix length to minimizing predicted prefill cost. The router calculates a unified cost metric that accounts for cache location, recompute overhead, and ongoing transfer congestion. To balance cache affinity and cluster load, the router identifies a candidate pool of workers whose estimated costs fall within a tunable threshold of the global minimum, and selects the lightest-loaded instance. To scale this efficiently, we deploy a two-tier routing framework: inter-node routing relies on lightweight historical snapshots to minimize cross-node network overhead, while intra-node routing utilizes real-time cache probing across fast CPU interconnects to avoid local PCIe transfer bottlenecks.

Second, a *batch-matching prefix cache reuse algorithm* reduces GPU stalls by dynamically partitioning prefix data at the worker level. Unlike prior work that relies on fixed recompute-transmission ratios [17, 24], CacheSched treats the predicted execution time of the currently running GPU batch as a strict, non-preemptible transfer budget. Using the performance model, it estimates the maximum number of prefix tokens that can be transferred within this window. The batch constructor then greedily populates the upcoming batch by loading only the prefix segments that fit within this budget, offloading the remainder to be seamlessly recomputed by the GPU. This dynamic partitioning aims to keep data prefetching within the predicted computation window.

We implement CacheSched on top of SGLang [66] and its standalone router module [2]. Extensive evaluations across synthetic and real-world conversational workloads on a multi-node testbed (up to 64 GPUs) demonstrate that CacheSched improves overall service throughput by up to 2.58× compared to state-of-the-art systems, while strictly adhering to practical time-to-first-token (TTFT) service-level objectives (SLOs).

2 Motivation

2.1 Prefix Caching in Multi-Worker Deployments

The substantial per-token memory footprint of the KV cache in modern LLMs severely constrains available GPU memory. For Llama-3.1-70B [1] using bfloat16 precision, the

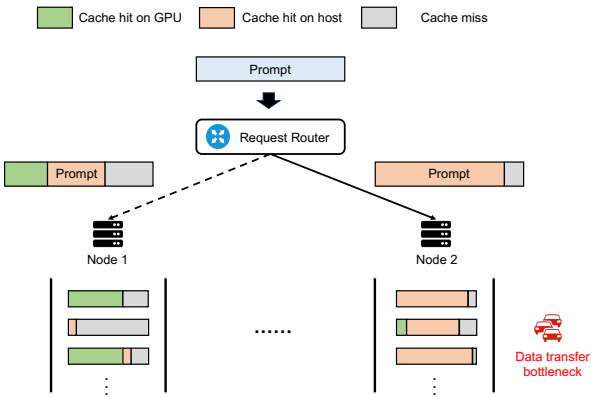


Figure 2. An illustrative example of inefficient inference request routing.

KV cache for a single token requires approximately 320 KB, and the cumulative memory required by many concurrent requests can quickly exhaust a GPU's capacity. Prefix caching addresses this by reusing KV cache entries across requests that share a common prefix, significantly reducing redundant prefill computation. To further expand cache capacity, production systems offload less-active prefixes to host DRAM [24, 56, 59, 66], creating a hierarchical cache spanning GPU memory and host memory. Recent hardware and software advances, including higher-generation PCIe interconnects and optimized transfer schemes [53], are steadily improving the efficiency of loading cached data from host memory back to the GPU.

In production, serving systems scale to multiple GPU workers. Each worker maintains its own subset of cached prefixes, and the system must decide which worker should serve each incoming request. This routing decision directly determines cache reuse efficiency and load distribution across the cluster. At the same time, within each worker, how much prefix cache to load from host memory versus recompute on the GPU also affects per-request latency. We analyze both problems below.

2.2 Limitations of Existing Request Routing

Existing routing strategies [46, 66] primarily rely on prefix match length to select a target worker. We refer to this class of strategies as longest prefix match routing. While longest prefix match maximizes cache hits in theory, it ignores the actual cost of serving a request on each worker, leading to two compounding problems.

First, longest prefix match is oblivious to cache location heterogeneity. It treats all cache hits as equivalent, failing to distinguish between data cached in GPU memory versus host memory. In practice, the cost of reusing a prefix from GPU memory is negligible, while reusing one from host memory requires a PCIe transfer that is significantly slower. In Figure 2, consider an incoming request where Worker 2 has the longest prefix match, but its entire cache resides in host

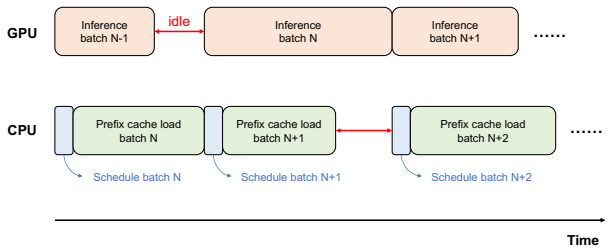


Figure 3. An illustrative example of GPU pipeline stalls in LLM inference.

memory with significant ongoing transfer contention. Worker 1 offers a shorter prefix match but has part of its cached data readily available in GPU memory. The longest prefix match algorithm would select Worker 2, but the resulting transfer bottleneck would block inference and cause severe delays. Routing to Worker 1, despite its shorter match, would have been a better choice.

Second, these per-request misjudgments accumulate into systematic load imbalance. Because longest prefix match greedily routes requests to the worker holding the longest matching prefix, subsequent requests from the same session are repeatedly directed to the same worker in multi-turn conversation workloads. Over time, a subset of workers accumulates disproportionately many active sessions while others remain underutilized. Under high concurrency, one or two overloaded workers dominate the tail latency of the entire cluster, limiting overall throughput even when aggregate resources are sufficient.

2.3 Inefficiency in Worker-Level Cache Loading

Beyond routing, there is also room for optimization within each worker. Modern inference engines asynchronously load prefix caches from host memory while the GPU executes the current batch. As shown in Figure 3, if the data transfer takes longer than the batch execution, the GPU stalls waiting for the next batch's data, wasting computational resources. Existing systems [43, 56, 66] attempt to maximize cache reuse by fetching the entire matched prefix, regardless of its size or location in the memory hierarchy. While this maximizes token reuse, it can trigger long transfers that stall the GPU pipeline. In our experiments, such stalls under SGLang's [66] full-prefix loading with HiCache [15] amount to 9.8%–27.4% of GPU prefill computation time (§5.4.1).

A more effective approach is partial loading: the worker dynamically divides the matched prefix into a portion to be transferred and a remainder to be recomputed by the GPU. By overlapping computation with transfer, partial loading can avoid pipeline stalls. However, choosing such a split point requires predicting both the batch execution time and the transfer time.

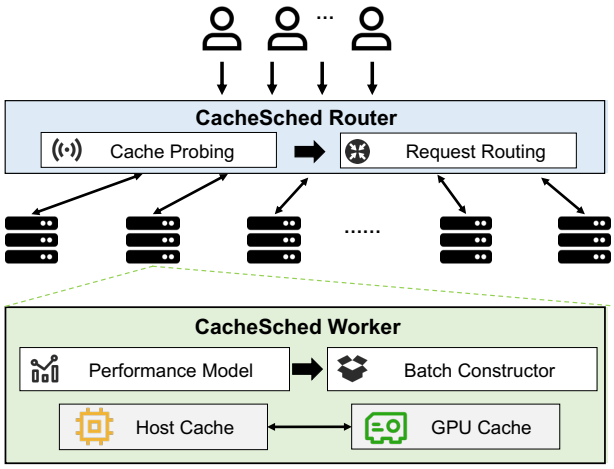


Figure 4. CacheSched overview.

2.4 Opportunity: Predictable Inference Performance

Both challenges above share a common requirement: cost estimation. The router needs to predict how long it would take each candidate worker to serve a given request; the worker needs to predict how much prefix cache can be transferred within the current batch's execution window.

A key enabler is that, unlike many traditional server workloads where latency depends on data-dependent control flow or contention from co-located tasks, LLM inference exhibits highly predictable performance characteristics. The deterministic structure of the transformer architecture means that, given the batch composition and model configuration, both the GPU execution time and the host-to-GPU transfer time can be estimated, making cost-aware scheduling feasible at both the routing and worker levels.

3 System Design

3.1 System Overview

Figure 4 presents the overall architecture of CacheSched, a distributed LLM serving system designed to maximize the efficient utilization of hierarchical prefix caches. As identified in §2, two performance problems arise in stateful LLM serving: suboptimal request distribution that ignores cache location heterogeneity and creates load imbalance, and GPU stalls caused by uncoordinated host-to-device data transfer. To address these, CacheSched is composed of two components built on top of a shared analytic performance model (§3.2): the CacheSched Router for cluster-level request dispatching, and the CacheSched Worker for local batch construction.

- **CacheSched Router (§3.4).** The CacheSched Router acts as the load balancer for the cluster, dispatching each incoming request to the optimal worker. It employs a heterogeneity-adaptive routing algorithm that makes decisions based on predicted prefill cost rather than match length alone. Its cache probing module queries workers to determine the availability and location (GPU memory or host DRAM) of a required prefix. This information is then combined with the performance model to estimate the actual prefill cost on each candidate worker. Unlike existing routers that greedily pursue the longest prefix match, our algorithm selects from a pool of cost-efficient workers and breaks ties by current load, balancing cache reuse with load distribution across the cluster.
- **CacheSched Worker (§3.3).** The CacheSched Worker is the inference engine responsible for batched request execution, with a design centered on preventing the host-to-device data transfer from becoming a bottleneck. Using the same performance model, it predicts the execution time of the current GPU batch and treats this time as a transfer budget. The batch constructor then loads only the amount of prefix cache from host DRAM that fits within this budget, and lets the GPU recompute the rest. This dynamic partitioning aims to overlap data transfer with computation and reduce GPU stalls.

The two components are connected through a continuous feedback loop. Workers periodically report their cache state and load metrics to the router, enabling it to make informed, globally-aware routing decisions that adapt to the dynamic state of the system.

Table 1. Symbol table for CacheSched's performance model.

Notation	Description
L_{seq}	Length of input sequence
$L_{uncached}$	Number of uncached tokens
L_{load}	Maximum number of loadable tokens
S_{prefix}	Size of the prefix cache to be transferred
P_{model}	Model size
BW_{GPU_MEM}	Effective bandwidth of GPU memory
BW_{load}	Effective bandwidth of loading host cache
$FLOPS$	FLOPS of GPU
M_{KV}	Per-token size of KV cache
$T_{prefill}$	Execution time of prefill phase
T_{decode}	Execution time of decode phase
T_{load}	Time cost of loading host cache
T_{budget}	Time budget for a batch

3.2 Performance Model of LLM Inference

Due to the distinct performance bottlenecks of the prefill and decode phases in LLM inference [18, 70], we model the two phases separately for accuracy. Notations used in our design are defined in Table 1.

To enable the key scheduling decisions in CacheSched at both the routing and worker levels, we develop a lightweight analytic performance model. The model answers two questions: (1) How long will a given batch of requests take to

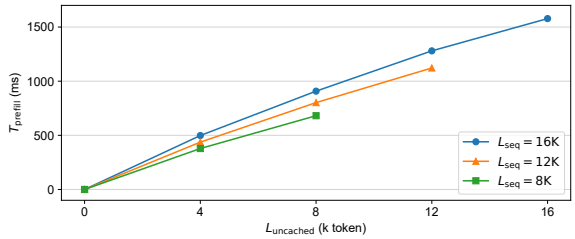


Figure 5. Empirical analysis for the prefill time.

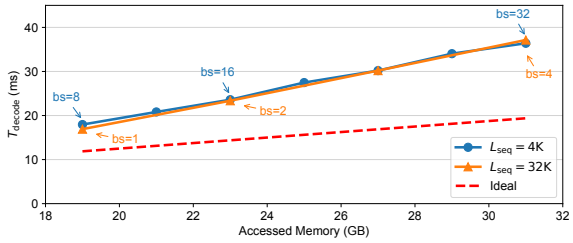


Figure 6. Empirical analysis for the decode time of a single decode iteration.

execute on the GPU? and (2) How long will it take to transfer a prefix cache of a given size from host memory? Answering these questions allows the router to estimate per-worker prefill cost, and allows the worker to compute the transfer budget for each batch.

3.2.1 Compute Batch Time Modeling. We first present the base models for the prefill and decode phases in turn, and then discuss how these models extend to tensor parallelism and chunked prefill.

Prefill Phase. The prefill phase processes a prompt's tokens in parallel and is characteristically compute-bound [18, 65]. Its latency is determined by the total number of floating-point operations (FLOPs) required. These operations primarily arise from two sources: attention computations, which scale with the sequence lengths, and model-weight-related computations like the feed-forward network (FFN) and projection matrices, which scale with the model size. For a batch containing requests with $L_{uncached}$ total new tokens and a maximum context length of L_{seq}, the total FLOPs can be expressed as approximately proportional to $(c_1 \times L_{seq} + c_2 \times P_{model}) \times L_{uncached}$ where c_1 and c_2 are constants. We approximate the execution time, $T_{prefill}$, by dividing this FLOPs estimate by the effective floating-point operations per second of the GPU, $FLOPS$:

$$T_{prefill} \approx \frac{(c_1 \times L_{seq} + c_2 \times P_{model}) \times L_{uncached}}{FLOPS} \approx (\alpha \times L_{seq} + \beta) \times L_{uncached} \quad (1)$$

At a fixed total length and with $u = L_{uncached}$, Equation 1 uses an attention term proportional to $L_{seq}u$, whereas the causal-attention work with a cached prefix is approximately $L_{seq}u - u^2/2$. The model omits the quadratic correction in u, replacing this concave dependence with a linear approximation in the number of uncached tokens.

Figure 5 plots the measured prefill time of Llama-3.1-8B [1] on an NVIDIA A100 40GB GPU, varying $L_{uncached}$ while holding L_{seq} at 8K, 12K, or 16K tokens. The measured curves are mildly sublinear in $L_{uncached}$; we approximate this dependence with a profiled per-token latency to keep the estimation lightweight:

$$T_{prefill} = \tau(L_{seq}) \times L_{uncached} \quad (2)$$

Here, $\tau(L_{seq})$ represents the amortized per-token processing latency in prefill phase, which encapsulates the model architecture and effective hardware performance and is obtained through offline profiling. The real-hardware prediction errors in §5.3.1 quantify the effect of this approximation on execution-time estimates.

Decode Phase. In contrast, the decode phase autoregressively generates one token per sequence in the batch. While the computation for a single token is minimal, it requires reading the entire set of model weights (P_{model}) and the KV cache for all sequences in the batch from GPU memory. Therefore, unlike prefill, this phase is memory-bound [18, 65, 70]. The execution time, T_{decode}, is dominated by the time taken to move this data.

The total data accessed from GPU memory is the sum of the model weights and the KV cache size, which is $P_{model} + \Sigma L_{seq} \times M_{KV}$ where ΣL_{seq} is the sum of sequence lengths in the batch. The decode time can thus be modeled as:

$$T_{decode} = \frac{P_{model} + \Sigma L_{seq} \times M_{KV}}{BW_{GPU_MEM}} \quad (3)$$

Critically, in our model we use the effective GPU memory bandwidth (BW_{GPU_MEM}), not the nominal hardware specification. Nominal bandwidth is an idealized peak value, whereas effective bandwidth is lower due to overheads like kernel launches and non-contiguous memory access patterns of attention mechanisms. To quantify this difference and validate our model, we measure the decode time across batch sizes (bs) from 1 to 32, while keeping the sequence length fixed within each batch.

Although larger batch sizes could theoretically increase the computational intensity, potentially making decode phase less memory-bound, our empirical results demonstrate that this effect is not significant within the evaluated range. Figure 6 plots the measured latency against the total accessed memory size. The results show a clear linear relationship, but the effective bandwidth, which represents the inverse of the slope, is significantly lower than what the nominal bandwidth would suggest. This linearity confirms that decode phase remains

memory-bound within the evaluated batch size range, and using an empirically profiled effective GPU memory bandwidth is essential for accurate decode time prediction.

Tensor parallelism (TP) and chunked prefill. When TP is used, model parameters are partitioned across multiple GPUs within a node, and each GPU processes the full sequence but only a shard of the model weights. The prefill phase remains compute-bound with the per-GPU FLOPs reduced proportionally, while the decode phase remains memory-bound with a smaller per-GPU model size. Inter-GPU communication (e.g., all-reduce) introduces additional overhead that scales with the TP degree. These overheads are implicitly captured by the profiled effective per-GPU compute throughput and memory bandwidth, which already reflect TP communication costs, without requiring structural changes to the model.

Additionally, under chunked prefill [18, 29] with hybrid batching, each iteration mixes prefill chunks with piggybacked decode tokens. We decompose such an iteration into a prefill-like compute component and a decode-like memory-access component, and the same per-phase models apply without structural changes. This decomposition does not explicitly model contention between the co-located prefill and decode work; §5.3.1 reports the resulting prediction error.

3.2.2 Prefix Cache Loading Time Modeling.

The second key component of our performance model is predicting the time required to load the prefix cache from host DRAM to GPU memory. This transfer is a prerequisite for reusing a cached prefix at host-side and can become a significant bottleneck. The loading time (T_{load}) follows a linear relationship, where it is the size of the prefix cache to be transferred (S_{prefix}) divided by the effective transfer rate (BW_{load}).

$$T_{load} = \frac{S_{prefix}}{BW_{load}} \tag{4}$$

It is also crucial to use the effective transfer rate rather than the nominal PCIe bandwidth specified by the hardware vendor. The nominal bandwidth is a theoretical peak that does not account for real-world overheads [53] such as kernel launch latency and fragmented data transmission caused by fine-grained page-based KV cache management. The empirically measured effective transfer rate can differ from the nominal specification, as Figure 7 shows. To improve our model's accuracy, CacheSched relies on a value for BW_{load} obtained through profiling that measures the actual throughput of transferring data blocks from host to GPU. Offline profiling measures prefill and decode times under controlled lengths and host-to-GPU loading concurrent with GPU execution (Figures 5–7); it takes about 10 minutes for each new model, GPU, and TP configuration, after which a single BW_{load} value is used for that configuration.

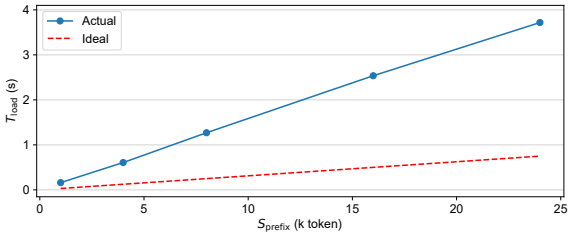


Figure 7. Empirical analysis for the loading time.

3.3 Batch-Matching Prefix Cache Reuse

Recent serving systems [43, 56, 66] attempt to maximize cache reuse by fetching the entire matched prefix for a request, regardless of its size or location in the memory hierarchy. While this strategy maximizes token reuse, it can trigger long-running data transfers from host DRAM over the PCIe bus. Because these DMA transfers are non-preemptible, if the data transfer takes longer than the execution of the current GPU batch, the GPU is forced into an idle state, creating a pipeline stall. To mitigate this critical inefficiency, CacheSched introduces a batch-matching prefix cache reuse mechanism. The core idea is to treat the execution time of the current GPU batch as a strict budget for data prefetching, dynamically partitioning the required prefix into a portion to be transferred and a remainder to be recomputed.

This dynamic partitioning requires the ability to predict both GPU compute time and data transfer time, which is the exact purpose of the analytic performance model detailed in §3.2. When constructing an upcoming prefill batch, CacheSched first predicts the execution time, T_{budget}, of the batch currently running on the GPU. This time T_{budget} becomes the budget for asynchronously transferring prefix caches from host memory for the new batch. Using the effective data transfer rate (BW_{load}) and per-token KV cache size (M_{KV}) derived from our model, the system calculates the maximum total number of prefix tokens, L_{load}, that can be transferred within this time window:

$$L_{load} = \frac{T_{budget} \times BW_{load}}{M_{KV}} \tag{5}$$

This L_{load} value represents the predicted total transfer capacity within the current batch's computation window. If the required host-resident prefix cache exceeds this capacity, the excess is recomputed on the GPU instead of being transferred. This budget is used to overlap prefetching with the current batch and avoid GPU stalls caused by prolonged transfers. Underestimating T_{budget} reduces the loading budget and can leave part of the actual computation window unused for cache loading, causing unnecessary recomputation. A modest overestimate is preferable here because loading can use the full actual window. Only the excess transfer beyond that window may remain exposed. Layer-wise asynchronous

Algorithm 1 Request scheduling of batch-matching prefix cache reuse

Input: Request waitlist W, maximum number of loadable tokens L_{load}
Output: Scheduled requests R
Initialize: $R \leftarrow []$
Sort Requests: Sort W by the number of cached tokens in descending order, but limit the loading portion by L_{load} (Linearly increase the priority of the longest-waiting request based on its queuing time first)
for each request r in W **do**
    if batch is full **then**
        Break: Stop scheduling
    Add r to R, and remove r from W
    Update L_{load} based on r
    Resort W
return R

loading can overlap part of this excess with subsequent computation. When loading is fast relative to computation, the budget usually covers the full matched prefix, so recomputation is triggered only for the portion whose transfer cannot be hidden.

The L_{load} capacity is used to guide the construction of the next prefill batch, as detailed in Algorithm 1. The algorithm greedily populates the batch to maximize cache reuse within the transfer budget. It begins by sorting requests in the waiting queue in descending order of their cached prefix length. It then iteratively selects requests for the batch, adding the length of their transferable prefix to a running total and ensuring this total never exceeds L_{load}. If a selected request's prefix is larger than the remaining capacity, only a portion is scheduled for transfer. After each selection, the remaining capacity is updated by subtracting the number of host-resident prefix tokens scheduled for transfer for that request, and priorities are re-evaluated to ensure the next choice is optimal given the new constraints. A batch is full when adding one more request would exceed its request-count, prefill-token (chunk) budget, or KV cache memory limit. GPU memory for the growing KV caches of running requests is handled by SGLang's existing memory manager, which pauses some decoding requests and returns them to the waiting queue when GPU memory becomes insufficient.

Figure 8 provides an illustrative example of this mechanism. A decode batch is currently executing on the GPU, and our model predicts its runtime, which translates into the transfer capacity L_{load} for the next batch. The scheduler first selects Request 4 (longest prefix), then re-evaluates and selects Request 1. Since Request 1's prefix exceeds the remaining budget, only part 1a is scheduled for transfer while 1b is marked for recomputation. With the budget fulfilled, the scheduler adds Request 3 (GPU cache only) to fill the batch. To avoid starvation of requests with shorter prefixes,

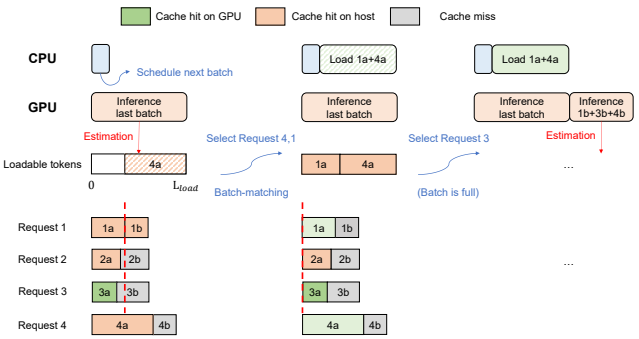


Figure 8. An illustrative example of batch-matching prefix cache reuse.

the scheduling priority is adjusted by a latency-aware factor that linearly increases with queuing time.

3.4 Heterogeneity-Adaptive Request Routing

While the batch-matching mechanism optimizes performance on a single worker, a practical serving system must scale to a multi-worker cluster. As analyzed in §2.2, routing strategies that rely primarily on prefix match length can create both per-request misjudgments and systematic load imbalance. To address this, CacheSched employs a cost-aware, heterogeneity-adaptive routing algorithm that balances prefix cache reuse with the estimated cost of data transfer and compute across the cluster.

3.4.1 Cost-Based Worker Selection. The central idea of our routing algorithm is to make decisions based on a unified cost metric: the predicted prefill latency for a given request on a potential worker. This metric naturally accounts for both recompute overhead (for uncached tokens) and data transfer penalties (for prefixes in host DRAM on a congested worker). To obtain this metric, the router's cache probing module queries each candidate worker to determine (1) whether a matching prefix exists, (2) where it resides in the memory hierarchy (GPU memory or host DRAM), and (3) the worker's current transfer load. The performance model from §3.2 then converts this information into a predicted prefill cost.

However, always dispatching requests to the single worker with the absolute lowest estimated cost can overload it while other workers remain idle. To mitigate this, our algorithm, outlined in Algorithm 2, uses a tunable threshold θ. It first identifies the minimum possible cost, $min_prefill_cost$, among all available workers, and then creates a candidate set of all workers whose cost is within a factor of θ of this minimum (i.e., $cost_i \leq \theta \times min_prefill_cost$). From this candidate set, it selects the worker with the lightest load (i.e., shortest request queue), balancing near-optimal cache reuse with load distribution.

Algorithm 2 Heterogeneity-adaptive request routing

Input: Request r, workers W, threshold ratio θ
Output: CacheSched worker w
Initialize: $best_workers \leftarrow []$, $min_prefill_cost \leftarrow \infty$, remove overloaded workers from W
for each worker w_i in W **do**
    Query Prefix Cache: Get cache status c_i of r for w_i
    Calculate Prefill Cost: Compute the prefill time cost $cost_i$ of r for w_i using c_i
    if $cost_i < min_prefill_cost$ **then**
        $min_prefill_cost \leftarrow cost_i$
for each worker w_i in W **do**
    if $cost_i \leq \theta \times min_prefill_cost$ **then**
        $best_workers.append(w_i)$
Select Worker: Choose the $best_worker$ from candidate set $best_workers$ with the lowest current load
return $best_worker$

θ controls the trade-off between cache affinity and load balance. When $\theta = 1.0$, the algorithm strictly selects the lowest-cost worker, maximizing cache reuse but risking load concentration. As θ increases, the candidate pool widens, giving more weight to load balance at the expense of cache optimality. We use $\theta = 1.1$ in all experiments without workload-specific tuning; replaying the routing decisions of the experiments in Table 3, varying θ from 1.05 to 1.5 changes only about 1% of placements, whereas $\theta = 1.0$ changes about 10%.

3.4.2 Two-Tier Routing Framework. The routing logic above is applied within a hierarchical two-tier framework that navigates the trade-off between information precision and communication overhead. In a centralized router, querying the real-time cache state of a remote worker provides accurate information for routing but incurs several milliseconds of communication latency including software and synchronization overhead. Conversely, using historical data is instantaneous but less accurate. Our two-tier design addresses this by using different strategies for routing among workers across physical nodes (inter-node) versus within a single node (intra-node).

Inter-node routing. It selects the best physical node in the cluster. To reduce costly cross-node network traffic, it uses a lightweight, historical cache state to approximate the prefill cost, defined simply as the number of uncached tokens. Workers periodically report their cache state summaries to the router, and the router makes decisions based on the latest available snapshot. This tier prioritizes low-overhead, approximate decisions.

Intra-node routing. It selects the best CacheSched worker within a node. Since communication between processes on a single node is much faster (via CPU interconnects), this tier queries the real-time cache status from each local worker.

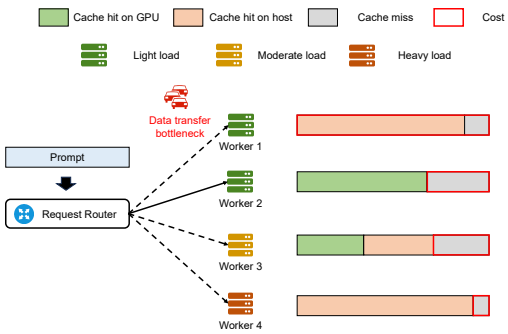


Figure 9. An illustrative example of the intra-node routing.

Its cost model is more sophisticated: if a worker's data transfer capability is fully occupied, the cost of loading a prefix from host DRAM is treated as equivalent to a full recompute, reflecting the potential for a GPU stall. This ensures the intra-node router avoids sending requests to workers that are transfer-congested.

The synergy between the router and the worker-level batch-matching is maintained through a feedback loop. Each worker periodically reports its cache state and current load to the router. Meanwhile, the batch-matching mechanism at each worker helps keep the actual prefill cost close to what the router predicted, because the worker dynamically adjusts its split between cache loading and recomputation to reduce stalls. This alignment between routing predictions and worker-level execution is what makes the two-tier system effective.

Figure 9 illustrates intra-node routing. When a new request arrives, the router first filters out overloaded workers like Worker 4. It then queries the real-time status of the remaining workers. Worker 1 has the longest prefix match, but it has planned a large amount of data transfer, which imposes a heavy cost penalty. Worker 3 offers the lowest prefill cost, while Worker 2 has a slightly higher cost. Assuming both fall within the candidate pool defined by θ, the router selects Worker 2, which has a lighter existing load, demonstrating a better balance between cache efficiency and system load.

4 Implementation

CacheSched's implementation[1] is based on SGLang [66] v0.4.5 (57f99608), a state-of-the-art LLM serving system that supports RadixAttention for prefix cache and HiCache [15] for hierarchical KV caching. We reuse some components of SGLang and its standalone request router, SGLang Router [2], then extend them to support CacheSched. The inter-node router is built upon SGLang Router to select physical node in the cluster, while the intra-node router is implemented in the data parallel (DP) controller of SGLang, responsible for performing real-time cache queries across DP instances.

[1] https://zenodo.org/records/23205571

5 Evaluation

5.1 Experiment Setup

Testbed. Our main experiments use a g6e.48xlarge instance, equipped with 8× NVIDIA L40S 48GB GPUs interconnected via PCIe 4.0, 192 vCPUs, and 1536GB of host memory. To further evaluate CacheSched on different hardware platforms, we use (1) a cluster of 8× g5.48xlarge instances, each configured with 8× NVIDIA A10G 24GB GPUs linked via PCIe 4.0 (64 GPUs total, for scalability evaluation), and (2) a server with 8× NVIDIA A100 SXM4 80GB GPUs (connected via NVLink).

Models. Our main experiments evaluate two representative open-source dense LLMs: Llama-3.1-8B-Instruct [1], one of the most widely adopted models for lightweight inference, and DeepSeek-R1-Distill-Qwen-14B [27], a distilled medium-parameter model that retains high performance while utilizing more GPU memory. In the extended evaluation, we additionally include Mixtral-8x7B-Instruct [33], a 46B-parameter mixture-of-experts (MoE) model.

Baselines. We select open-source LLM serving systems that provide both request routing and inference execution as our main baselines, based on the following two criteria: (1) the ability to asynchronously load and evict KV cache layer-by-layer between host and GPU memory, and (2) the use of prefix-cache-aware request routing. Therefore, we choose SGLang [66] and the vLLM Production Stack [16] as our main baselines. In our evaluation, both systems are configured with a number of runtime replicas equal to the number of GPUs. They are allocated identical and large enough memory budgets (both GPU and host memory) for KV cache management. Chunked prefill is enabled for all systems including CacheSched. We do not enable prefill-decode (PD) disaggregation. We additionally compare against Preble [46] as a reference for cache-aware request routing. We do not directly evaluate Dynamo [12] or Mooncake [43]; as discussed in §7, their routers trade off prefix reuse against worker load in the same way as SGLang Router, and the component ablation in §5.4.1 isolates CacheSched's routing gain over this baseline.

- **SGLang(v0.4.5, 57f99608)**: SGLang is a state-of-the-art LLM inference system that supports prefix caching through RadixAttention and hierarchical caching through HiCache [15]. Its routing component, SGLang Router [2], dispatches requests to SGLang runtimes according to its cache-aware load-balancing algorithm. When the load is imbalanced (i.e., the difference between the most and least loaded nodes exceeds a threshold), it selects the node with the fewest requests. Otherwise, it selects the node with the longest prefix match.
- **vLLM Production Stack(0.1.5, 0b6a61c)**: This is a production-grade distributed deployment framework built upon the foundational vLLM [38] engine, known for its PagedAttention mechanism. It supports asynchronous layer-by-layer KV cache offloading and loading via LMCache [10]. Its prefix-aware router selects the node with the longest prefix match.

Table 2. Key Characteristics of Two Workloads.

Workload	W1	W2
Avg. Length		
New Prompt Input	512	96
Output	96	348
Ratio	5.333:1.000	1.000:3.625
Conversation Turns	10	≥ 4

Workloads. The workloads used in the experiments include the SGLang multi-turn conversation benchmark **(W1)** [66] and the ShareGPT4 dataset **(W2)** [3]. The SGLang multi-turn conversation benchmark allows for controlled testing with predefined patterns, ensuring a comprehensive evaluation of performance under controlled conditions. The ShareGPT4 dataset is a real-world conversational dataset containing user-assistant conversation traces collected from a large corpus, capturing diverse prompt lengths and reasoning complexity. Requests for both datasets are set to arrive according to a Poisson distribution, enabling the simulation of stochastic arrival processes commonly encountered in real-world systems. Table 2 presents the key statistics of the two workloads.

It is crucial to note that for each new request in a multi-turn conversation, the actual input comprises the entire conversational history (all previous inputs and outputs) plus the new user prompt. Consequently, as the number of turns increases, the context length grows, leading to a significant increase in the ratio of prefill length to decode length per request.

Metrics. Our evaluation focuses on two key latency metrics TTFT (Time to First Token) and TBT (Time between Tokens), which capture different aspects of system performance. TTFT measures the complete processing duration from request arrival to the first token generation, including both the queuing and prefill phases. As this metric directly impacts user-perceived responsiveness, it serves as our primary indicator for our performance enhancements. TBT quantifies the average inter-token generation latency, excluding the initial token. We include this metric to illustrate that CacheSched maintains decoding efficiency with minimal overhead.

5.2 Overall Performance

We report the average TTFT and TBT of CacheSched on both the SGLang multi-turn conversation benchmark and ShareGPT4 dataset on a single AWS EC2 g6e.48xlarge instance equipped with 8× NVIDIA L40S 48GB GPUs, with a focus on throughput optimization under a practical TTFT service-level objective (SLO). Similar to prior works [26, 50], we set the TTFT SLO to 1 second.

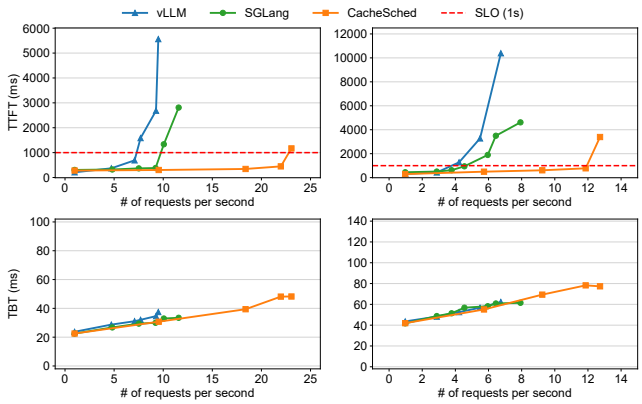


Figure 10. Overall performance on the SGLang multi-turn conversation benchmark. (a) Llama-3.1-8B-Instruct. (b) DeepSeek-R1-Distill-Qwen-14B.

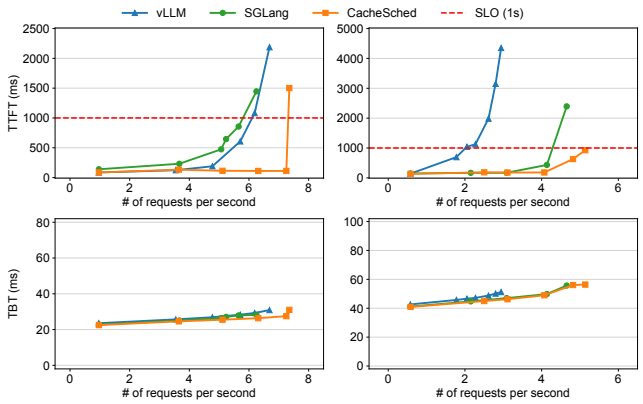


Figure 11. Overall performance on the ShareGPT4 trace. (a) Llama-3.1-8B-Instruct. (b) DeepSeek-R1-Distill-Qwen-14B.

TTFT. As shown in Figure 10 and 11, CacheSched achieves throughput improvements under the TTFT SLO. Experimental results reveal that CacheSched improves request throughput by up to 1.26× over SGLang and 2.55× over vLLM Production Stack on ShareGPT4. For the multi-turn conversation benchmark, the improvements are more substantial, reaching up to 2.58× and 3.16× over SGLang and vLLM Production Stack respectively. CacheSched does not introduce extra overhead under light load.

In both workloads, SGLang and vLLM Production Stack show comparable performance, and both suffer from varying degrees of load imbalance. While load is partially balanced across most instances, one or two instances consistently handle a disproportionately high load, becoming performance bottlenecks and degrading overall throughput. This observation directly reflects the limitation of existing routing strategies we analyzed earlier: by greedily directing requests based on prefix match length, they concentrate active sessions on a few workers. CacheSched's heterogeneity-adaptive routing avoids this by balancing cache affinity with load distribution. Interestingly, on the ShareGPT4 dataset with the DeepSeek-R1-Distill-Qwen-14B model, the overload phenomenon for SGLang is somewhat alleviated, contributing to its relative performance recovery compared to the multi-turn conversation benchmark.

The multi-turn conversation benchmark has a prefill length that is significantly longer than the decode length, a characteristic more pronounced than in ShareGPT4. This allows CacheSched to more effectively leverage its cache-aware scheduler with less queuing, thereby achieving better performance.

TBT. Notably, the TTFT improvements of CacheSched do not come at the cost of degraded decoding efficiency. When operating at comparable throughput levels to the baselines, the TBT of CacheSched is statistically indistinguishable from SGLang and vLLM Production Stack. This is because the design of CacheSched does not change the decoding process or increase the overhead at the scheduler. Even at higher throughput beyond the baselines' capabilities, CacheSched exhibits only a modest increase in TBT, ensuring minimal impact on the user experience during the decoding phase.

5.3 Generalizability and Scalability

We further evaluate CacheSched on different hardware platforms to validate its generalizability and scalability. We did not include vLLM Production Stack in this comparison as its performance is even worse than that of SGLang in the previous experiments.

5.3.1 Hardware and Model Generalizability. We evaluate CacheSched on 8× NVIDIA A100 SXM4 80GB GPUs, using both Llama-3.1-8B-Instruct and Mixtral-8x7B-Instruct with the multi-turn conversation benchmark. As a reference, we include Preble [46], a prefix-cache-aware routing system also built on SGLang. Preble routes each request by a binary exploit-or-explore decision: it sends the request to the GPU caching the longest matched prefix when cache-hit tokens outnumber uncached ones, or to the least-loaded GPU otherwise. Beyond the throughput under the 1s TTFT SLO, we also report *goodput* [67], defined as the effective request rate at which requests jointly satisfy both the TTFT and TBT SLOs. We set the TTFT SLO to 1 s for both models, the TBT SLO to 30 ms for Llama and 60 ms for Mixtral, and report goodput at 90% joint SLO attainment.

As shown in Table 3, despite also being built upon SGLang, Preble achieves performance close to or below SGLang on

Table 3. Throughput and goodput on 8×A100.

	Llama-3.1-8B			Mixtral-8x7B (TP=2)		
	SGLang	Preble	CacheSched	SGLang	Preble	CacheSched
Throughput (req/s)	10.72	7.63	28.03	6.03	6.73	15.26
vs. SGLang	1.00×	0.71×	2.61×	1.00×	1.12×	2.53×
Goodput (req/s)	6.33	3.93	16.90	3.72	3.76	10.04
vs. SGLang	1.00×	0.62×	2.67×	1.00×	1.01×	2.70×

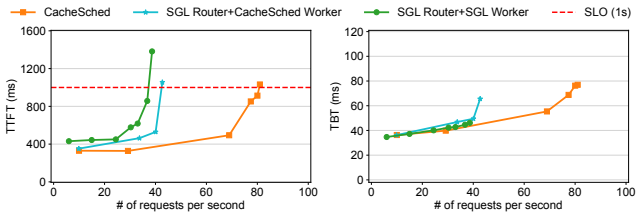


Figure 12. Overall performance on a cluster of 64 GPUs on the SGLang multi-turn conversation benchmark.

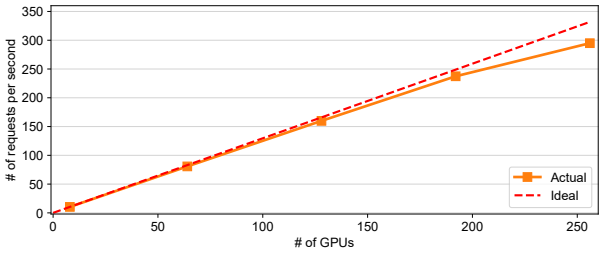


Figure 13. Routing scalability in simulation, compared with ideal linear scaling from the 8-GPU baseline.

both models. In contrast, CacheSched achieves 2.61× throughput over SGLang on Llama-3.1-8B and 2.53× on Mixtral-8x7B under the 1s TTFT SLO, confirming that the improvement generalizes to MoE architectures. Against Preble, the throughput improvements are 3.67× and 2.27×, respectively. Under the stricter goodput metric, the gap widens further: CacheSched achieves 2.67× and 2.70× over SGLang on Llama and Mixtral respectively, and up to 4.30× over Preble.

We also examine the performance model's prediction errors in these experiments. For Llama-3.1-8B, the P90 absolute error in predicting one decode iteration is 6.3 ms, equivalent to 8.3% of the median prefill-batch execution time, and these decode predictions tend to overestimate execution time. As analyzed in §3.3, this error direction is preferable for batch matching when the overestimate is modest, allowing cache loading to use the full actual overlap window. With chunked prefill, the combined prefill and decode model predicts iteration time with a P90 relative error of 15.9% compared with the measured execution time, indicating moderate error when the phase models are applied jointly. Predictions of host-to-GPU loading time have a P90 relative error below 20% under normal contention. These estimates guide routing preferences and the division of cached prefixes between loading and recomputation, but exact latency predictions are not required for these decisions.

5.3.2 Scalability. On a cluster of 64 GPUs (8× g5.48xlarge instances, each with 8× NVIDIA A10G 24GB GPUs), we evaluate the scalability of CacheSched using Llama-3.1-8B-Instruct with multi-turn conversation benchmark. We compare CacheSched with SGLang and a mixed setup. The mixed setup refers to running CacheSched on each node locally, with inter-node routing managed by SGLang Router.

As shown in Figure 12, CacheSched outperforms both SGLang and the mixed setup under the 1s TTFT SLO. Specifically, CacheSched achieves 2.17× request throughput over SGLang, and 1.90× over the mixed setup. The gap between full CacheSched and the mixed setup shows that cross-node routing becomes increasingly important in multi-node deployments.

To study throughput scaling beyond our physical testbed, we use a simulator that replays request traces through CacheSched's actual routing logic. It assigns each GPU computation the execution time predicted by the analytical model and varies only the GPU count. The limited prediction errors observed on real hardware support using these estimates to assess whether throughput scales nearly linearly with node count. This simulation is intended only to study scalability beyond the scale of our real-hardware experiments, not to replace them.

Figure 13 compares the simulation results with an ideal linear-scaling reference normalized to the 8-GPU baseline. CacheSched almost scales linearly, reaching 89% of ideal throughput at 256 GPUs. The remaining gap is mainly due to less precise cross-node state information at larger scales.

5.4 Ablation Study

5.4.1 Component Contribution. To isolate the contribution of each component, we incrementally add CacheSched's components to the SGLang baseline using the experiments from §5.3. We evaluate three configurations: SGLang (baseline), SGLang Router with CacheSched Worker (+Batch-Matching), and full CacheSched (+CacheSched Router). The latter two configurations use the same CacheSched Worker,

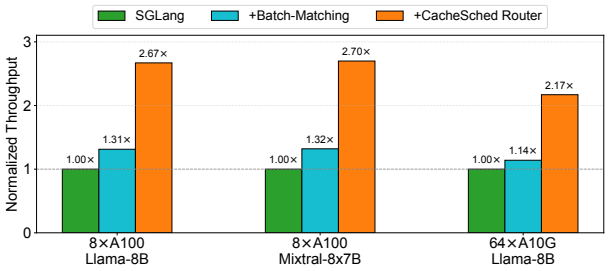


Figure 14. Component contribution of CacheSched. Throughput is normalized to SGLang under 1s TTFT SLO.

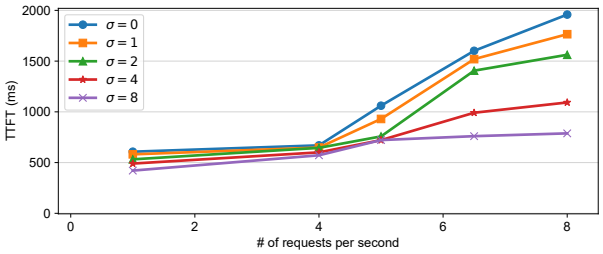


Figure 15. Sensitivity analysis by varying request skewness.

so their comparison isolates the contribution of the router. Figure 14 shows the throughput under the 1s TTFT SLO, normalized to SGLang, across three setups before: 8×A100 with Llama-3.1-8B, 8×A100 with Mixtral-8x7B, and 64×A10G with Llama-3.1-8B.

Across the three setups, SGLang Router with CacheSched's batch-matching worker achieves normalized throughput of 1.31×, 1.32×, and 1.14×. Replacing SGLang Router with CacheSched Router increases these values to 2.67×, 2.70×, and 2.17×, respectively. Routing therefore provides an additional 1.90–2.05× throughput over the same worker with SGLang's cache-aware router, accounting for approximately 81%–88% of the total improvement over SGLang. The gains come primarily from routing decisions rather than simply adding hierarchical-cache support or worker-side batch-matching.

To explain the benefit of batch matching at the worker, we examine GPU stalls caused by prefix cache transfers. Under the full prefix loading policy used by SGLang with HiCache, the ratio of this stall time to GPU prefill computation time ranges from 9.8% to 27.4% across the evaluated conditions. CacheSched uses the current batch's computation time as the budget for prefix cache loading. Batch matching limits transfers to this budget to avoid these GPU stalls.

5.4.2 Routing Overhead. We measure the overhead introduced by CacheSched's routing decisions. The cache probing latency is approximately 4 ms on average (3 ms median), and the end-to-end scheduling path takes approximately 5 ms median. Compared to the prefill execution time, which typically ranges from tens to hundreds of milliseconds, the routing overhead is negligible.

5.4.3 Request Skewness and Routing Effectiveness. To evaluate how CacheSched performs under workloads with different levels of request patterns, we conduct a sensitivity analysis by varying the skewness of request round counts. In this experiment, the number of conversation rounds per request follows a distribution with a fixed mean of 10, while the standard deviation (σ) of the distribution is adjusted from 0 to 8 to represent increasing skewness. A larger σ indicates greater variability in request complexity, that is, more extreme differences between short and long conversations.

As shown in Figure 15, CacheSched maintains stable performance across different levels of skewness. As σ increases, CacheSched exhibits no performance degradation, demonstrating its robustness across different workload intensities. We further examine the routing decisions by analyzing the top-k worker selection distribution. Across all skewness levels, CacheSched routes over 90% of requests to the top-1 worker by cache-hit potential, while a least-load baseline only achieves 34% to 48%. This indicates that the heterogeneity-adaptive routing consistently makes cache-efficient decisions even under diverse workload patterns.

6 Discussion

Prefill-decode (PD) disaggregation. CacheSched is designed and evaluated under PD-colocated serving, which remains prevalent in production due to its simpler operation and lower networking requirements [61].

Nevertheless, the proposed design is also compatible with PD disaggregation deployments [31, 42, 67], as its core of independently modeling prefill and decode phases directly aligns with the resource specialization of PD disaggregation. The performance model needs no structural changes, and the batch-matching prefix reuse algorithm retains its core logic for prefill scheduling. In a disaggregated setup, the routing component would operate over the prefill worker pool, where cache-aware placement remains applicable.

MoE models and expert parallelism. Our evaluation includes the MoE model Mixtral-8x7B-Instruct [33], demonstrating that CacheSched's performance model and scheduling algorithms generalize to MoE models under tensor parallelism and data parallelism.

However, expert parallelism (EP) deployment [19, 64, 71] introduces additional routing considerations. EP requires synchronizing attention computation timing across workers to form batches for FFN processing among all experts. This demand for cross-worker synchronization conflicts with the current routing logic, which independently assigns requests to workers. Adapting CacheSched to EP would require the performance model to further distinguish between attention

and FFN operations, and the router to account for EP-specific load balancing constraints.

Sparse and linear attention. CacheSched's performance model and batch-matching mechanism rely on the standard multi-head attention architecture, where KV cache size scales linearly with sequence length. Sparse attention designs reduce the memory-to-compute ratio during prefill [52], diminishing the data transfer bottleneck that batch-matching targets. Similarly, linear attention variants replace quadratic attention with linear recurrences, further shifting this ratio. Under these architectures, the transfer-budget formulation would need to be re-calibrated. The heterogeneity-adaptive routing, however, remains applicable as long as prefix cache reuse provides a latency benefit over recomputation.

7 Related Work

KV cache management. KV cache is a core component in LLM inference, as it stores intermediate attention results to avoid redundant computations. Numerous recent studies [35, 38, 49, 56, 63] have explored various approaches to optimize KV cache utilization and management. Prefix caching [25, 57, 66] improves cache reuse across requests. Offloading the KV cache to host memory or storage [23, 24, 39, 41, 52–55, 59] increases the effective cache capacity. Pensieve [59] preferentially retains later token chunks in GPU or host memory. It restores chunks from host memory layer by layer and recomputes discarded chunks at the beginning of the sequence. HCache [24] stores hidden states and partitions layers using offline hardware profiling. It uses this partition to combine restoration of hidden states with token recomputation or KV cache offloading. In both systems, the recovery path is determined in advance by cache residency or hardware profiles. CacheSched instead adapts the division between cache loading and recomputation to each batch's computation window. This mechanism is designed jointly with request routing.

Llumnix [47] monitors instance-level memory imbalance and reactively migrates running requests from one instance to another, transferring the request's entire accumulated KV cache to the destination. To reduce migration downtime, it overlaps the transfer with ongoing decoding by exploiting the append-only property of KV cache. In contrast, CacheSched avoids imbalance proactively through heterogeneity-adaptive routing before execution. After routing, CacheSched further decides how much of the matched prefix in host memory to restore and how much to recompute under a transfer budget, rather than always transferring the full KV state.

Request routing. Efficient workload placement [22, 40, 60, 69] and communication [11, 36, 68] are key research problems in cluster-level scheduling. Cache-aware routers [30, 46, 48, 51, 61] balance prefix reuse against worker load. SGLang Router [2] chooses between the worker with the longest matching prefix and the worker with the shortest request queue. Dynamo [12] jointly considers the prefill computation for uncached tokens and the KV cache load of active requests. Without cache pooling or migration, Mooncake [43] combines prefill time with queueing time, accounting for prefix reuse in the prefill cost. Its full scheduler also accounts for KV cache transfer time when a prefix is remote. When one factor dominates, all three favor either greater prefix reuse or lower worker load. Their differences lie in the granularity of cost estimates and how the two factors are combined. The SGLang Router baseline in our evaluation already covers this shared tradeoff between prefix reuse and worker load.

Since prefix reuse costs depend on the cache's location in the memory hierarchy and the current load on the transfer path, CacheSched incorporates both factors into routing. It coordinates these routing decisions with batch matching at the worker, where cache loading is constrained by a transfer budget.

Worker-level request scheduling. Within an inference engine, request scheduling [21, 28, 34, 37, 58, 65, 70] targets efficient utilization of resources within the hardware constraints. Sarathi-Serve [18] and FastGen [29] combine prefill and decode in a batch to balance the need for computation and memory access. Additionally, some studies on scheduling focus on fairness [45] or SLO adherence [32, 62] in multi-user LLM serving. In CacheSched, we greedily populate each batch to maximize cache reuse within the transfer budget and reduce GPU stalls.

8 Conclusion

We propose CacheSched, a scheduling system tailored for multi-worker LLM serving with hierarchical prefix caching. Driven by a lightweight analytic performance model, it integrates heterogeneity-adaptive request routing algorithm and batch-matching prefix cache reuse algorithm, for request routing across cluster and request scheduling within inference engine, respectively. Evaluations on both synthetic and real-world workloads show that CacheSched significantly improves the performance over state-of-the-art LLM inference systems, achieving up to 2.58× overall throughput under a practical TTFT SLO.

Acknowledgments

This work was supported in part by the Scientific Research Innovation Capability Support Project for Young Faculty under the grant number ZYGXQNJSKYCXNLZCXM-I1, the National Natural Science Foundation of China under the grant number 62172008, the Fundamental Research Funds for the Central Universities, Peking University, and the ZTE Industry-University-Institute Cooperation Funds under Grant No.IA20240827028. Rilin Huang and Xin Jin are also affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] 2024. Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/.

[2] 2024. SGLang Router. https://lmsys.org/blog/2024-12-04-sglang-v0-4/.

[3] 2024. Sharegpt4 Dataset. https://huggingface.co/datasets/openchat/openchat_sharegpt4_dataset.

[4] 2025. ChatGPT. https://openai.com/chatgpt/overview/.

[5] 2025. Claude. https://claude.ai/.

[6] 2025. Cursor. https://cursor.com/.

[7] 2025. DeepSeek. https://chat.deepseek.com/.

[8] 2025. Gemini Deep Research. https://gemini.google/overview/deep-research/.

[9] 2025. LightLLM. https://www.light-ai.top/lightllm-blog/.

[10] 2025. LMCache. https://docs.lmcache.ai/.

[11] 2025. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[12] 2025. NVIDIA Dynamo KVBM. https://docs.nvidia.com/dynamo/v-0-8-1/components/kvbm/architecture.

[13] 2025. NVIDIA TensorRT-LLM. https://docs.nvidia.com/tensorrt-llm/index.html.

[14] 2025. OpenAI Deep Research. https://openai.com/index/introducing-deep-research/.

[15] 2025. SGLang HiCache. https://lmsys.org/blog/2025-09-10-sglang-hicache/.

[16] 2025. vLLM Production Stack. https://docs.vllm.ai/projects/production-stack/en/latest/index.html.

[17] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. INFERCEPT: efficient intercept support for augmented large language model inference. In *Proceedings of the 41st International Conference on Machine Learning*. Article 3, 15 pages.

[18] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[19] DeepSeek-AI. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[20] Letian Deng and Yanru Zhao. 2023. Deep learning-based semantic feature extraction: A literature review and future directions. *ZTE communications* 21, 2 (2023), 11.

[21] Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. 2025. PrefillOnly: An Inference Engine for Prefill-only Workloads in Large Language Model Applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. 399–414.

[22] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024. MuxServe: flexible spatial-temporal multiplexing for multiple LLM serving. In *Proceedings of the 41st International Conference on Machine Learning*. Article 473, 13 pages.

[23] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 111–126.

[24] Shiwei Gao, Youmin Chen, and Jiwu Shu. 2025. Fast State Restoration in LLM Serving with HCache. In *Proceedings of the 20th European Conference on Computer Systems*. 128–143.

[25] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine learning and systems* (2024).

[26] Ruihao Gong, Shihao Bai, Siyu Wu, Yunqian Fan, Zaijun Wang, Xiuhong Li, Hailong Yang, and Xianglong Liu. 2025. Past-Future Scheduler for LLM Serving under SLA Guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 798–813.

[27] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081 (Sept. 2025), 633–638. doi:10.1038/s41586-025-09422-z

[28] Jiaao He and Jidong Zhai. 2024. FastDecode: High-Throughput GPU-Efficient LLM Serving using Heterogeneous Pipelines. arXiv:2403.11421 [cs.DC] https://arxiv.org/abs/2403.11421

[29] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. arXiv:2401.08671 [cs.PF] https://arxiv.org/abs/2401.08671

[30] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. arXiv:2406.17565 [cs.DC] https://arxiv.org/abs/2406.17565

[31] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. arXiv:2401.11181 [cs.DC] https://arxiv.org/abs/2401.11181

[32] Jinqi Huang, Yi Xiong, Xuebing Yu, Wenjie Huang, Entong Li, Li Zeng, and Xin Chen. 2025. SLO-Aware Scheduling for Large Language

Model Inferences. arXiv:2504.14966 [cs.DC] https://arxiv.org/abs/2504.14966

[33] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[34] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. 2024. NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference. arXiv:2411.01142 [cs.DC] https://arxiv.org/abs/2411.01142

[35] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. arXiv:2404.12457 [cs.DC] https://arxiv.org/abs/2404.12457

[36] Xin Jin, Zhen Zhang, Yunshan Jia, Yun Ma, and Xuanzhe Liu. 2024. SDCC: Software-defined collective communication for distributed training. *Science China Information Sciences* 67, 9 (2024), 192104.

[37] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. *POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference*. 897–912.

[38] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[39] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. 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)*. 155–172.

[40] Xuanzhe Liu, Yihao Zhao, Shufan Liu, Xiang Li, Yibo Zhu, Xin Liu, and Xin Jin. 2024. MuxFlow: efficient GPU sharing in production-level clusters with more than 10000 GPUs. *Science China Information Sciences* 67, 12 (2024), 222101.

[41] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. arXiv:2507.07400 [cs.DC] https://arxiv.org/abs/2507.07400

[42] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2025. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*. 118–132.

[43] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*. 155–170.

[44] Tianqi Ren, Rongpeng Li, Mingmin Zhao, Xianfu Chen, Guangyi Liu, Yang Yang, Zhifeng Zhao, and Honggang Zhang. 2025. Separate Source Channel Coding Is Still What You Need: An LLM-Based Rethinking. *ZTE COMMUNICATIONS* 23, 1 (2025).

[45] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 965–988.

[46] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2025. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. In *The 13th International Conference on Learning Representations*.

[47] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 173–191.

[48] The AIBrix Team, Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, Yifei Zhang, Yiqing Zhu, Shuowei Jin, Gangmuk Lim, Binbin Chen, Zuzhi Chen, Xiao Liu, Xin Chen, Kante Yin, Chak-Pong Chung, Chenyu Jiang, Yicheng Lu, Jianjun Chen, Caixue Lin, Wu Xiang, Rui Shi, and Liguang Xie. 2025. AIBrix: Towards Scalable, Cost-Effective Large Language Model Inference Infrastructure. arXiv:2504.03648 [cs.DC] https://arxiv.org/abs/2504.03648

[49] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. arXiv:2506.02634 [cs.DC] https://arxiv.org/abs/2506.02634

[50] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 640–654.

[51] Bingyang Wu, Zili Zhang, Yinmin Zhong, Guanzhe Huang, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2025. TokenLake: A Unified Segment-level Prefix Cache Pool for Fine-grained Elastic Long-Context LLM Serving. arXiv:2508.17219 [cs.DC] https://arxiv.org/abs/2508.17219

[52] Yongtong Wu, Shaoyuan Chen, Yinmin Zhong, Rilin Huang, Yixuan Tan, Wentao Zhang, Liyue Zhang, Shangyan Zhou, Yuxuan Liu, Shunfeng Zhou, Mingxing Zhang, Xin Jin, and Panpan Huang. 2026. DualPath: Breaking the Storage Bandwidth Bottleneck in Agentic LLM Inference. arXiv:2602.21548 [cs.DC] https://arxiv.org/abs/2602.21548

[53] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. 2025. Strata: Hierarchical Context Caching for Long Context Language Model Serving. arXiv:2508.18572 [cs.DC] https://arxiv.org/abs/2508.18572

[54] Yi Xiong, Hao Wu, Changxu Shao, Ziqing Wang, Rui Zhang, Yuhong Guo, Junping Zhao, Ke Zhang, and Zhenxuan Pan. 2024. LayerKV: Optimizing Large Language Model Serving with Layer-wise KV Cache Management. arXiv:2410.00428 [cs.DC] https://arxiv.org/abs/2410.00428

[55] Yi Xu, Ziming Mao, Xiangxi Mo, Shu Liu, and Ion Stoica. 2024. Pie: Pooling CPU Memory for LLM Inference. arXiv:2411.09317 [cs.LG] https://arxiv.org/abs/2411.09317

[56] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the 20th European Conference on Computer Systems*. 94–109.

[57] Lu Ye, Ze Tao, Yong Huang, and Yang Li. 2024. ChunkAttention: Efficient Self-Attention with Prefix-Aware KV Cache and Two-Phase Partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 11608–11620.

[58] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[59] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful Large Language Model Serving with Pensieve. In *Proceedings of the 20th European Conference on Computer Systems*. 144–158.

[60] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. 2025.

Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving. arXiv:2505.04021 [cs.DC] https://arxiv.org/abs/2505.04021

[61] Dingyan Zhang, Jinbo Han, Kaixi Zhang, Xingda Wei, Sijie Shen, Chenguang Fang, Wenyuan Yu, Jingren Zhou, and Rong Chen. 2026. Simple is Better: Multiplication May Be All You Need for LLM Request Scheduling. arXiv:2603.15202 [cs.DC] https://arxiv.org/abs/2603.15202

[62] Wei Zhang, Zhiyu Wu, Yi Mu, Banruo Liu, Myungjin Lee, and Fan Lai. 2025. Tempo: Application-aware LLM Serving with Mixed SLO Requirements. arXiv:2504.20068 [cs.DC] https://arxiv.org/abs/2504.20068

[63] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*. Article 1506, 50 pages.

[64] Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. 2025. DeepEP: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP.

[65] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yang Zhou, Jiarong Xing, and Ion Stoica. 2024. BlendServe: Optimizing Offline Inference for Auto-regressive Large Models with Resource-aware Batching. arXiv:2411.16102 [cs.LG] https://arxiv.org/abs/2411.16102

[66] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2025. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. Article 2000, 27 pages.

[67] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

[68] Yang Zhou, Zhongjie Chen, Ziming Mao, ChonLam Lao, Shuo Yang, Pravein Govindan Kannan, Jiaqi Gao, Yilong Zhao, Yongji Wu, Kaichao You, et al. 2025. An Extensible Software Transport Layer for GPU Networking. arXiv:2504.17307 [cs.DC] https://arxiv.org/abs/2504.17307

[69] Yiheng Zhou, Wei Zeng, Qingfang Zheng, Zhilong Liu, and Jianping Chen. 2024. A Survey on Task Scheduling of CPU-GPU Heterogeneous Cluster. *ZTE Communications* 22, 3 (2024), 83.

[70] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2025. NanoFlow: Towards Optimal Large Language Model Serving Throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 749–765.

[71] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. 2025. MegaScale-Infer: Efficient Mixture-of-Experts Model Serving with Disaggregated Expert Parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 592–608.