Requests of a Feather Must Flock Together: Batch Size vs. Prefix Homogeneity in LLM Inference

Saksham Rathi
Indian Institute of Technology Bombay
Mumbai, India
sakshamrathi@cse.iitb.ac.in

Preeti
Indian Institute of Technology Bombay
Mumbai, India
preeti@cse.iitb.ac.in

Mythili Vutukuru
Indian Institute of Technology Bombay
Mumbai, India
mythili@cse.iitb.ac.in

Abstract

Auto-regressive token generation in large language models is memory-bound because it requires "attending to" key and value tensors (KV cache) of all previous tokens. Prior work aims to improve the efficiency of this *decode* process by batching multiple requests together, and maximizing batch size subject to GPU memory constraints. The key observation of our work is that with prefix-sharing workloads, smaller, prefix-homogeneous batches – where *all* requests share a common prefix – can achieve higher decode throughput than larger, heterogeneous batches, due to better spatial and temporal locality during KV cache accesses. However, current prefix-aware schedulers maximize prefix reuse within a batch only to reduce KV cache memory footprint, but do not stop batch formation at smaller homogeneous batches that could have performed better. Further, we show that shared prefix detection in existing schedulers relies on radix-tree traversals, incurring substantial CPU overhead that is often comparable to GPU execution time. This paper presents Feather, a prefix-aware scheduler that uses reinforcement learning (RL) to learn the optimal tradeoff between batch size and prefix homogeneity. We also introduce Chunked Hash Tree (CHT), a lightweight data structure that enables fast prefix detection and efficient request selection for the RL scheduler, avoiding expensive tree traversals. We integrate Feather into vLLM and SGLang, and our evaluation shows that Feather achieves 2–10× higher end-to-end throughput as compared to existing schedulers by reducing the total number of KV cache accesses, while doing no worse than the status quo when there is insufficient prefix sharing.

CCS Concepts: • **Computing methodologies → Machine learning**.

Keywords: LLM Inference; Prefix Sharing; Batching and Scheduling; Reinforcement Learning



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/10.1145/3828523.3857087

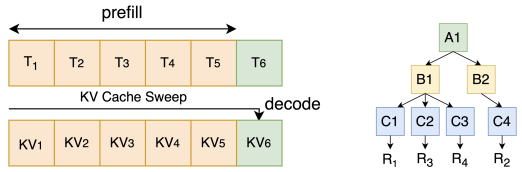
ACM Reference Format:
Saksham Rathi, Preeti, and Mythili Vutukuru. 2026. Requests of a Feather Must Flock Together: Batch Size vs. Prefix Homogeneity in LLM Inference. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3828523.3857087

1 Introduction

Large language models (LLMs) [12, 26, 38] are rapidly transforming modern computing, powering applications such as conversational assistants [10, 13], code generation [15], and retrieval-augmented generation [18, 41]. As these models transition from research to production, cloud AI engines [4, 14, 25] must improve serving latency and resource efficiency for both online and offline inference. While prior work has proposed scheduling strategies [1, 28, 47, 49], parallelization techniques [34, 39], memory management methods [17, 31, 44], quantization [9, 20], and kernel-level optimizations [27, 43], efficiently serving LLM workloads, especially under realistic high-throughput conditions, remains an open problem.

An LLM inference request goes through two phases: *prefill* and *decode*. The prefill phase processes all prompt tokens in parallel to produce the first output token, making it compute-bound. The decode phase, by contrast, generates tokens auto-regressively and is fundamentally memory-bound. Each newly generated token attends to the keys and values of all previous tokens, resulting in a sweep of the entire KV cache for each decode iteration, as shown in Figure 1(a). As context lengths grow, this memory traffic becomes the dominant bottleneck. Popular LLM inference engines like vLLM [17] adopt batching to improve GPU utilization during the decode phase, scheduling multiple requests together in a batch using policies like First Come First Serve (FCFS). More advanced frameworks explore techniques such as interleaving [1], disaggregation [49], and spatial co-scheduling [16, 21], along with dynamically adjusting batch sizes based on runtime constraints [28], to address this compute-memory mismatch.

Orthogonally, workloads such as few-shot prompting [8, 30], system prompts [35], and RAG [18] exhibit significant *prefix sharing* across requests. For instance, when an LLM agent repeatedly analyzes a massive codebase or a document repository, the same background is used as a prefix to the prompt given in every sub-task. As context lengths



(a) Prefill and Decode **(b)** Radix Tree

Figure 1. LLM Inference Primitives

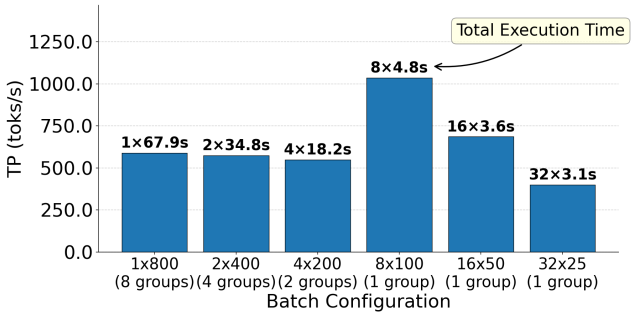


Figure 2. Batch Size vs. Prefix Homogeneity

grow to millions of tokens, optimizing for these shared prefix workloads becomes critical to achieving good inference performance. Existing work primarily relies on reusing KV caches for shared prefixes, to eliminate redundant KV computation and reduce KV cache memory footprint. Systems such as SGLang [46] and vLLM [17] use radix trees (Figure 1(b)) or hashing-based structures to keep track of shared KV caches. While vLLM's FCFS would have scheduled requests in arrival order (R_1, R_2, R_3, R_4), SGLang performs a depth-first traversal of the radix tree and schedules requests in the order R_1, R_3, R_4, R_2, to maximize KV cache reuse and reduce KV cache evictions. A complementary line of work [27, 40, 42, 43] exploits the shared prefix structure within the attention kernel itself, by jointly computing attention for groups of queries that share prefixes. Such prefix-aware attention kernels avoid redundant loading of shared KV blocks for each request separately, thereby reducing memory traffic.

Despite these advances, we identify two critical gaps in current scheduling strategies for prefix-shared workloads. First, prior work focuses on maximizing batch size as the primary knob to improve token generation throughput in the decode phase. However, we show via controlled experiments that prefix homogeneity—the extent to which *all* requests in a batch share a common prefix—has an even stronger impact on performance. Further, we find that the performance gains increase with the length of the prefix shared across all the requests in a batch (§3.1). This is because when a prefix is shared by all requests, there is significant *spatial* and *temporal* locality in the memory accesses of the shared KV cache located in the GPU DRAM. Such highly regular and sequential memory access patterns allow hardware prefetchers to detect uniform strides, and proactively move data from DRAM to the L2 cache. These benefits of long sequential memory access patterns to GPU DRAM have been documented in prior work as well [22, 29], though not in the context of prefix-homogeneous batches. We show that these locality benefits eventually translate to both higher memory bandwidth utilization and lower total amount of data fetched from DRAM, and hence higher decode throughput.

We also show that moderately small prefix-homogeneous batches can leverage these locality benefits and perform better than larger heterogeneous batches (§3.2). For example, Figure 2 shows the throughput and execution times of 800 requests, where 8 groups of 100 requests each share a large prefix of 10K tokens between them. Processing these requests in 8 prefix-homogeneous batches of 100 requests each leads to nearly 2× higher throughput than when using larger heterogeneous batches, though very small homogeneous batches (32 batches of 25 requests each) perform poorly as well due to sub-optimal compute utilization. Existing prefix-aware schedulers have no mechanism to recognize this tradeoff between batch size and prefix homogeneity; e.g., in Figure 1(b), a DFS-traversal-based scheduler will use R_2 to maximize batch size, even if a smaller batch (only R_1, R_3, R_4) could have performed better due to a longer common prefix.

The second big gap with current prefix-aware schedulers is that they incur prohibitive CPU overheads in maintaining and traversing radix trees, which can account for 50–90% of total latency (§3.3). Even though radix tree based schedulers optimize the GPU execution time through better reuse of KV caches as compared to simpler schedulers like FCFS, we show that the additional overheads on the critical execution path to detect prefix sharing, all done while the GPU is waiting, result in an overall worse end-to-end performance.

This paper introduces Feather[1] (§4), a prefix-aware scheduler that improves throughput by efficiently finding an optimal balance between batch size and prefix homogeneity. Feather is built on two components: *(i) Chunked Hash Tree (CHT)*, a lightweight data structure for fast prefix detection, similar in spirit to vLLM's block table [17]. Instead of performing token-level radix tree traversals, CHT represents each request as a vector of cumulative prefix hashes. To efficiently select a request that maximizes the length of the prefix shared across the active batch, CHT maintains a *min-heap*, updated lazily, and ordered by per-request *missing count*—the number of prefix chunks not yet covered by the batch. This enables the identification of the best candidate in $O(\log W)$ time among W waiting requests, a significant improvement over DFS traversal [46]. *(ii) RL-based batching policy.* Deciding when to stop adding requests to a batch presents a trade-off: larger batches improve utilization but dilute prefix homogeneity. We cast this as a reinforcement

[1]Please see [32] for an explanation of this name.

learning problem in which the scheduler observes batch characteristics (e.g., current size, shared prefix length), evaluates the impact of adding another request to the batch, and stops batch formation at a point that maximizes throughput. This adaptive approach consistently outperforms static heuristics across diverse workloads and hardware configurations. Together, these two components enable Feather to construct prefix-homogeneous batches efficiently, both improving system throughput and reducing scheduling overhead.

We implement Feather on vLLM [17] and SGLang [46] and evaluate it across diverse workloads and configurations (§5). Our results show that Feather achieves 2–10× higher end-to-end throughput over vLLM's FCFS and other baselines for shared prefix lengths ranging from 1K to 10K tokens, while also improving DRAM bandwidth utilization. Feather is fully hardware-agnostic and yet performs better than recent prefix-aware kernels that must be tuned to specific hardware configurations. Our RL policy ensures that our performance never falls below that of the baselines, even if the workload does not allow for prefix-homogeneous batches, and CHT reduces prefix-detection overhead by up to 1000× compared to DFS-based approaches for long sequences.

In summary, this paper makes the following contributions: (i) We demonstrate the tradeoff between batch size and prefix homogeneity in the decode phase performance of LLM inference, and identify previously overlooked CPU overheads in prefix-aware scheduling. (ii) We design the Chunked Hash Tree (CHT) data structure for low-overhead prefix detection, and develop a reinforcement learning-based policy for adaptive batch construction. (iii) We build Feather, a prefix-aware scheduler integrated into vLLM and SGLang, and demonstrate improvements in throughput and latency on diverse workloads. Our code is available at https://github.com/networkedsystemsIITB/Feather.

2 Background and Related Work

We now discuss the background and prior work on batching, scheduling, and prefix-aware optimizations.

2.1 Background

LLM Inference and Attention. LLM inference proceeds auto-regressively in two phases: prefill and decode. Their latencies are known as *time to first token* (TTFT) and *time between tokens* (TBT), respectively. Each decode step comprises self-attention, Query-Key-Value-Output (QKVO) projections, and Multi-Layer Perceptron (MLP) layers. Self-attention captures global token dependencies as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$$

where d denotes the head dimension. During the decode phase, the query $\mathbf{Q}$ of the current token attends to all prior keys $\mathbf{K}$ and values $\mathbf{V}$. These are cached in GPU memory as the **KV cache**, eliminating redundant computation at the cost of increased memory usage. The asymmetry between prefill and decode, together with the growing memory footprint of the KV cache, motivates much of the prior work.

GPU Execution Model. GPUs are hierarchically organized for parallelism across hundreds of thousands of threads. The basic compute unit is the Streaming Multiprocessor (SM), and modern GPUs typically contain a few hundred SMs. Each SM includes an L1 cache, shared memory, tensor cores for matrix multiplication (GEMM), and execution units. SMs access global memory through a shared L2 cache. This hierarchy, from small, fast on-chip memory to large, high-latency global memory, is critical for LLM serving, as KV cache access patterns directly influence memory bandwidth utilization. Like CPUs, GPUs use hardware prefetching and benefit from regular, sequential memory access patterns [22, 29].

2.2 Related Work

Batching and Scheduling. vLLM [17] adopts a first-come, first-served (FCFS) scheduling policy as a practical baseline for LLM serving. Sarathi-Serve [1] reduces prefill-decode interference through *chunked prefills* and stall-free batching, while DistServe [49] eliminates it by disaggregating the two phases across GPUs. At the kernel level, POD-Attention [16] overlaps prefill and decode on a single GPU via fused attention with SM-aware scheduling, and Bullet [21] enables fine-grained resource sharing through dynamic SM allocation. At the request level, BucketServe [47] groups requests by sequence length to improve batching efficiency, and Pang et al. [28] dynamically adapts the batch size to runtime constraints. Complementary to prior work, *Feather operates at the scheduler level and learns when to stop batch construction based on prefix homogeneity, introducing a scheduling signal not explicitly exploited in existing systems.*

Prefix Sharing. Many real-world workloads [8, 18, 30, 35] exhibit substantial prefix overlap, and enabling KV cache reuse avoids redundant computation and reduces memory footprint. However, realizing these benefits requires coordination between serving and memory management. SGLang [46] enables prefix sharing via *RadixAttention*, which uses a radix tree to organize token sequences, and enables KV reuse across shared prefixes. Similarly, vLLM [17] implements prefix caching [36] using a Merkle-tree-inspired hashing scheme (Figure 3), where each KV block hash depends on its tokens and preceding prefix hashes, enabling reuse when hash sequences match. BatchLLM [48] targets large offline workloads by grouping prefix-sharing requests and reordering them to maximize KV reuse. BlendServe [45] combines a resource-aware prefix tree with scheduling to jointly optimize prefix reuse and compute–memory demand. *Feather uses chunked hashing instead of radix trees or hash chains, for more efficient prefix-homogeneous batch construction.*

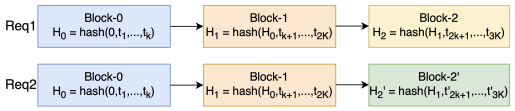


Figure 3. An Example of a vLLM Merkle Tree

Prefix-Aware Attention Kernels. Prior work [27, 40, 42, 43] also optimizes attention kernels by grouping queries with shared prefixes to load KV blocks once and reuse them across queries. These approaches, however, rely on GPU-specific kernel designs. In contrast, *Feather improves inputs to the existing attention kernels by constructing prefix-homogeneous batches upfront, making it hardware-agnostic and naturally composable with kernel-level techniques.*

3 Motivation

Through a set of controlled experiments[2] (here and in [32]), we validate three hypotheses that underpin the design of Feather: (i) prefix homogeneity, the extent to which *all* requests in a batch share a common prefix, has a strong impact on the decode throughput because it increases the locality of KV cache accesses. Importantly, these gains can be achieved *without any modifications to existing attention kernels.* (ii) There is a trade-off between batch size and prefix homogeneity, where moderately sized homogeneous batches *can* outperform large heterogeneous batches; and (iii) existing prefix detection approaches incur substantial CPU overhead.

Experimental Setup. All experiments use the Llama 3 8B [2] model and are run on an RTX 6000 Ada GPU [23] with 48 GB of GDDR6 memory and a 96 MB L2 cache. We use the vLLM [17] inference engine with a default maximum batch size of 500 unless stated otherwise. We first focus on decode throughput, measured in output tokens per second. To isolate decode behavior, we perform a warm-up run so that the prefix KV caches are resident in GPU memory. This ensures that per-token compute remains constant within an experiment and differences arise primarily from memory accesses and batching. We also report end-to-end throughput, which includes the time taken to prefill the queries. For this model, a 10k-token request generates 1.2 GB of KV cache data across all layers. Within each layer, vLLM uses its paged memory management system and allocates the KV cache in blocks of 16 tokens contiguously in GPU memory, which is sufficient for locality benefits to kick in. We sample requests from the L-Eval [3] dataset and prune the size of each request as required by the experiment. We use the same FlashAttention [11] backend (FlashAttention-2, via vLLM 0.13.0) throughout all the experiments, so that observed throughput differences are attributable only to batch composition rather than kernel-level variation.

[2]For additional content, we refer the reader to the appendices in the extended version of our paper on arXiv [32].

3.1 Significance of Prefix Homogeneity

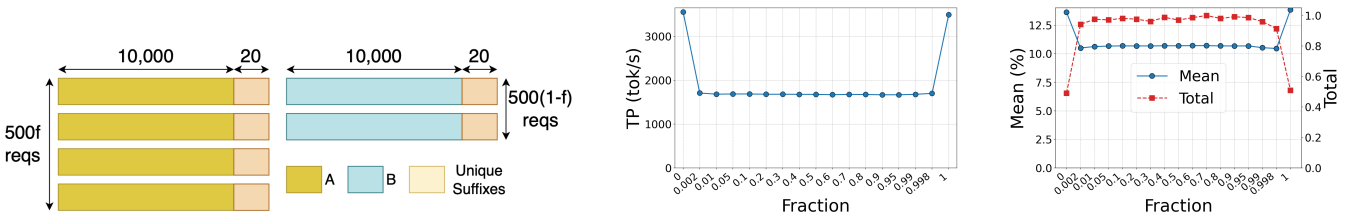
We start by observing the impact of prefix homogeneity on decode phase throughput. When *all* requests in a batch share a common prefix, their attention operations traverse the same KV cache regions concurrently. This yields two beneficial effects: (i) spatial locality: Sequential access to contiguous KV tensors enables the hardware prefetcher to fetch upcoming cache lines from DRAM into L2 [22, 29]; (ii) temporal locality: Cache lines fetched for one request are reused by neighboring requests before eviction. Together, these effects increase L2 hit rates and DRAM utilization, improving throughput. In contrast, heterogeneous prefixes interleave KV accesses across disjoint regions, disrupting locality and reducing effective memory bandwidth. We will now present experiments that validate these claims.

Experiment 1: We consider two prefixes, A and B, each 10K tokens long. We form a batch of 500 requests, each using the prefix A or B, followed by a unique suffix of length 20 tokens (Figure 4(a)). Each request generates 50 decode tokens. Let $f \in [0, 1]$ denote the fraction of requests using prefix A. When f is 0 or 1, the batch is homogeneous; otherwise, two prefix groups coexist. We perform a warm-up run so that KV caches for both prefixes are resident in GPU memory.

Figure 4(b) shows decode throughput as a function of f. We see that the throughput is highest for fully homogeneous batches. However, even a single request from the other prefix group (e.g., $f = 0.002$) causes throughput to drop by nearly 2×. This demonstrates that decode throughput is sensitive to even minimal prefix heterogeneity. To understand this effect, we capture DRAM bandwidth utilization using DCGMI [24]. We first plot the mean bandwidth utilization for each f, and then we sum the utilization over the entire experiment (i.e., compute the area under the memory bandwidth curve) to approximate total memory accesses. As shown in Figure 4(c), homogeneous batches achieve over 40% higher bandwidth utilization than mixed-prefix batches due to greater spatial locality, enabling more efficient hardware-level prefetching into the GPU cache. Furthermore, greater temporal locality in prefix-homogeneous batches reduces the total data fetched from DRAM. Because most SMs process queries from the dominant prefix even when $f = 0.002$, we attribute the performance gains primarily to improved L2 cache reuse rather than increased L1 cache locality within an SM.

Takeaway 1: *Decode throughput is maximized when all requests share a prefix; even a single divergent prefix disrupts locality and sharply reduces effective bandwidth.*

Experiment 2: Next, we evaluate the impact of the shared prefix length on performance. For this, we construct a batch of 100 requests, each 2K tokens long and sharing a common prefix of length $2K \times f$ tokens, where $f \in [0, 1]$. The remaining $2K \times (1 - f)$ tokens are unique across requests (Figure 5). The KV caches fit into GPU memory for all values of f. Figure 6 shows that throughput increases smoothly



(a) Visualization **(b)** Throughput **(c)** Bandwidth Utilization

Figure 4. Prefix Homogeneity - Two Prefix Groups

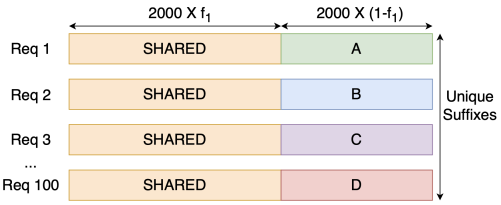


Figure 5. Fractional Sharing Visualization

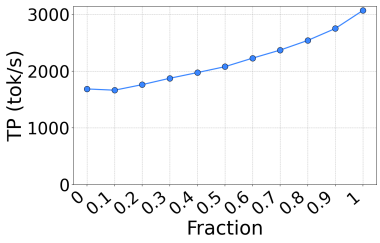


Figure 6. Throughput vs. Fraction of Prefix Shared

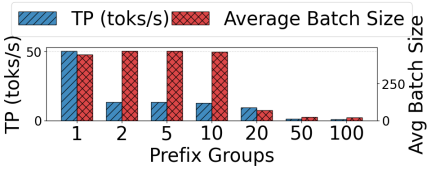


Figure 7. Number of Prefix Groups

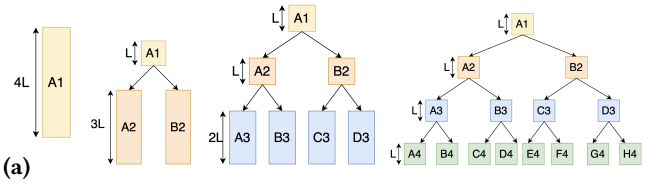
with f. It is lowest at $f = 0$ and improves steadily as sharing increases, reaching over 60% higher throughput at $f = 1$. As f grows, a larger portion of the KV cache is shared across requests; since decode repeatedly traverses the KV cache, greater spatial and temporal locality improves throughput.

Takeaway 2: *In prefix-homogeneous batches, decode throughput increases smoothly with the length of the shared prefix.*

Experiment 3: Next, we evaluate the effect of the number of prefix groups on performance using an experiment with N shared prefixes $P_1, P_2, \ldots, P_N$, each of length $10K$ tokens and having same number of requests. Figure 7 shows throughput and average batch size as N increases. As observed earlier, throughput drops from $N = 1$ to $N = 2$ due to reduced temporal and spatial locality arising from prefix heterogeneity. However, from $N = 2$ to $N = 20$, throughput remains stable, indicating that additional heterogeneity has a limited impact once locality is disrupted. Finally, throughput drops sharply from $N = 20$ to $N = 100$, as GPU memory can no longer hold KV caches for all active prefixes. The impact is also visible through the reduced average batch size for $N = 50$ and $N = 100$. Prior work [46, 48] mitigates the throughput drop from $N = 20$ to $N = 50$ due to KV cache evictions by maximizing prefix reuse and minimizing memory footprint using depth-first traversal of radix trees. However, to the best of our knowledge, no prior work has observed the drop in performance from $N = 1$ to $N = 2$ prefix groups.

Takeaway 3: *While prefix heterogeneity causes a drop in throughput, the amount of heterogeneity has a marginal impact on the loss of locality. However, very high heterogeneity leads to minimal KV reuse and higher GPU memory evictions.*

Experiment 4: Until now, all the experiments have considered requests that have a single level of prefix sharing.



(a) 1-level **(b)** 2 levels **(c)** 3 levels **(d)** 4 levels

Figure 8. Radix Tree Sharing Levels

We now ask the question: what is the impact of more complicated sharing patterns on performance? For this, we construct radix trees with 1 to 4 levels of shared prefixes (Figure 8). There are two branches in each level, and requests are equally distributed across the leaves. The sequence length per request is fixed at $4L$, and Figure 9 shows the decode throughput for three values of L. As expected, a single level of radix tree has the highest performance because of maximum locality. Interestingly, radix trees with 2, 3, and 4 levels exhibit similar throughput across different values of L, despite differing branching structures. Why so? During the initial segment of length L, all configurations access the same KV region. However, once branching occurs, the batch becomes heterogeneous, locality degrades, and the marginal cost of additional prefix groups is small, as seen earlier in Figure 7.

Takeaway 4: *Performance is driven mainly by the length of the top-level prefix shared by all requests, while the number of radix-tree levels and branches has a negligible impact.*

3.2 Batch Size vs Prefix Homogeneity

So far, we have established that prefix homogeneity has a significant impact on performance. But when the workload does not have enough requests from a prefix group and

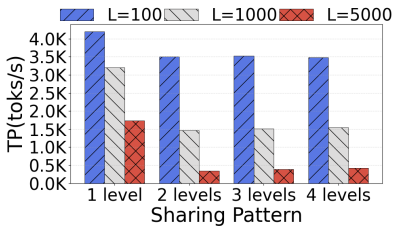


Figure 9. Radix Tree Sharing Patterns

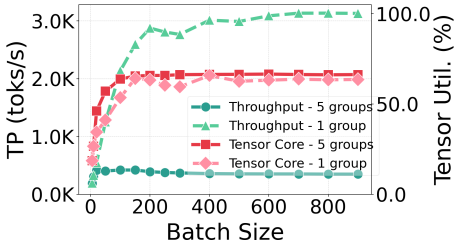


Figure 10. Throughput vs. Batch Size

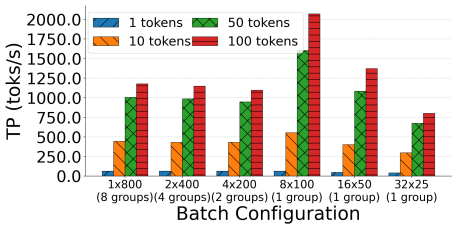


Figure 11. Batch Size vs. Homogeneity

prefix-homogeneous batches are too small, the compute will remain under-utilized, leading to a loss in throughput. Therefore, there exists a trade-off between batch size and prefix homogeneity. We now explore this trade-off.

Experiment 5: We study decode throughput vs. batch size using two workloads: (i) heterogeneous, with requests spread across 5 distinct 10K-token prefixes, and (ii) homogeneous, with all requests sharing one such prefix. Figure 10 shows throughput and tensor core utilization as a function of batch size for both workloads. We see that throughput plateaus at a point before the maximum batch size, and the saturation throughput is higher in the homogeneous case due to greater locality. Further, the batch size at which throughput saturates is higher for the homogeneous workload because requests can access the same KV cache without using additional memory bandwidth. Interestingly, the compute utilization is almost exactly the same across both workloads. Similar observations have also been made in prior work [33].

Takeaway 5: *Decode is memory bandwidth bound and increasing batch size beyond a point does not necessarily improve throughput. However, the point at which diminishing returns begin depends on prefix homogeneity (and hardware config).*

Experiment 6: Given that batch size has a diminishing impact on performance, we ask if moderately small homogeneous batches can outperform larger heterogeneous ones. To answer this, we use eight prefix groups with 100 requests each (800 total, each 10K tokens) and evaluate the following batching strategies: (i) 1×800 (8 prefix groups/batch); (ii) 2×400 (4 prefix groups/batch); (iii) 4×200 (2 prefix groups/batch); (iv) 8×100 (homogeneous); (v) 16×50 (homogeneous); and (vi) 32×25 (homogeneous). We vary the decode length from 1 to 100 tokens and measure the end-to-end throughput, including both the prefill and decode phases for all batches. Figure 11 shows this throughput across the different batching strategies. For small decode lengths, throughput is dominated by prefill, and large heterogeneous batches perform the best due to higher tensor core utilization; splitting 1×800 into smaller batches slightly reduces throughput due to lower compute efficiency, as locality benefits are not yet dominant. As decode length increases, performance trends reverse: multiple decode iterations repeatedly sweep the KV cache and reap greater benefits due to locality. Therefore, at longer decode lengths, fully homogeneous batches (8×100) significantly improve throughput despite smaller batch sizes. Notably, even 16×50 continues to outperform 1×800 at large decode lengths, showing that locality gains can outweigh reduced tensor core utilization. Only when batches become too small (32×25) does throughput decline again, as reduced arithmetic intensity and occupancy dominate.

Takeaway 6: *Moderately small prefix-homogeneous batches can achieve higher throughput than larger heterogeneous ones. Gains increase with longer decode lengths as repeated KV sweeps amplify locality; however, excessively small batches underutilize compute and degrade performance.*

Policy	GPU Exec (s)	Batch Time (s, %)	Throughput (toks/s)
vLLM FCFS	229.36	3e-6 (0.00%)	436.25
SGLANG DFS-W	177.78	92.15 (51.8%)	562.61
SGLANG LPM	149.27	46.62 (31.2%)	669.92
PAT (vLLM Block Table)	316.45	91.72 (29.0%)	316.63
vLLM RadixTree	139.33	62.95 (45.2%)	717.35
vLLM User IDs	105.43	0.03 (0.03%)	948.54

Table 1. Overhead of Dynamic Prefix Detection

3.3 Overhead of Dynamic Prefix Detection

Next, we study how state-of-the-art inference engines identify prefix sharing and the costs involved. Systems such as SGLang [46] and BatchLLM [48] do not explicitly form prefix-homogeneous batches, but use radix trees to detect sharing, and schedule requests to minimize KV recomputation.

Experiment 7: We construct a workload of 2K requests, each with a 10K-token prefix, and compare scheduling CPU time to total execution time. SGLang proposes two prefix-aware policies that use its radix tree: *Longest Prefix Match*, which prioritizes requests with the longest matching cached prefixes, and *DFS-Weight*, which performs a depth-first traversal and prioritizes branches with more active requests. vLLM uses a much simpler hash-based block table, which is used for grouping requests in the prefix-aware attention kernel implementation of PAT [43]. We compare the overheads of this block table, along with our own radix-tree-based scheduler implemented on top of vLLM. We also compare against an idealized policy where prefix group identifiers are provided explicitly, eliminating dynamic prefix discovery.

Table 1 shows that SGLang's LPM and DFS-Weight incur scheduler overhead (measured as time taken by all functions

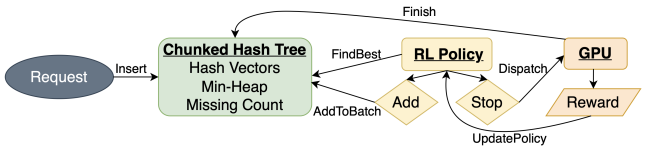


Figure 12. Feather Pipeline

invoked during batch formation) comparable to GPU execution time, even though these policies improve performance over vLLM's FCFS. In contrast, when prefix group identifiers are provided explicitly, CPU overhead drops significantly. This gap arises from the structure of radix trees: they require token-by-token comparisons during insertion, dynamic node splitting on partial matches, and repeated tree traversals for scheduling requests. With long prompts and high concurrency, these operations incur significant overhead. Similarly, vLLM's block table design is inefficient in this regard. However, these overheads are not sufficiently highlighted in prior work [45, 48], which is mostly based on offline inference.

Takeaway 7: *Existing approaches introduce significant CPU overhead when detecting prefix sharing, which is sometimes even comparable to the GPU execution time.*

4 Design and Implementation

Our experiments so far have demonstrated that the length of the prefix shared across *all* active requests is an important determinant of system performance. However, this metric interacts with batch size in a complex way: larger batches improve GPU utilization but often reduce shared prefix alignment. Further, detecting this shared prefix length incurs high overhead in existing systems. We now describe the design of Feather, an efficient scheduler that finds the right balance between batch size and prefix homogeneity.

4.1 Overview

Feather is based on the following two key ideas.

Key Idea 1: Learn the Stopping Decision via RL (§4.3). The trade-off between batch size and prefix homogeneity raises a key question: when should the scheduler stop adding requests? The answer depends on both hardware and workload, making fixed heuristics brittle. We model batch construction as a sequential decision process and learn a stopping policy via Reinforcement Learning. At each step, the scheduler evaluates a candidate request using various state features and decides whether it should be added to the batch or if the batch should be dispatched to the GPU for execution.

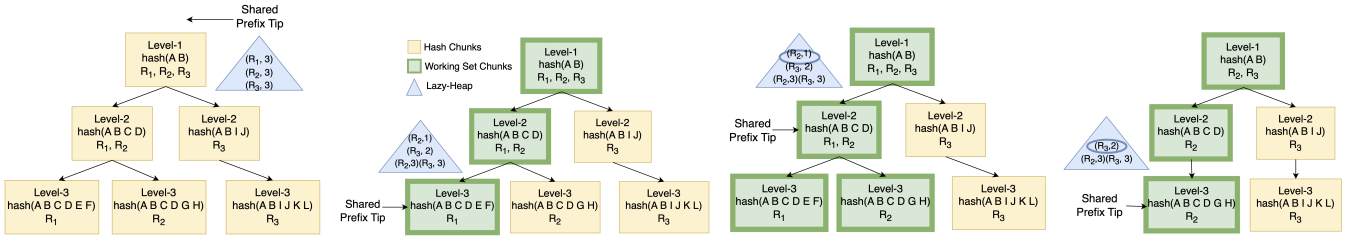
Key Idea 2: Shared Prefix Detection via Chunked Hash Tree (§4.2). Tree-based approaches incur significant overhead (§3.3), and finding the best waiting request requires a DFS traversal that is $O(E)$, where E is the number of edges in the radix tree. In the worst case, this can go up to $O(W \times T)$, where W is the number of waiting requests, and T is the sequence length. We instead represent each request as a sequence of hashes over fixed-size chunks and maintain a working set of hashes for the active batch. This is similar to Merkle-style hash chains but does not require sequential hash computation, and each chunk can be processed independently without prefix dependencies. Prefix matching is thus reduced to a set-overlap problem: selecting the request with the maximum overlap with the working set. However, a naive implementation that scans all requests and counts overlaps requires $O(W \cdot C)$ work per decision, where C is the number of chunks in a request, which remains too expensive for frequent scheduling. We avoid this by maintaining a missing count per request—the number of chunks absent from the working set—which changes only when the working set changes (i.e., upon request addition or completion). Overlaps are thus updated incrementally, reducing selection to finding the request with the smallest missing count. To do so efficiently, we maintain a min-heap over missing counts, with lazy updates, yielding $O(\log W)$ selection cost.

System Architecture. Figure 12 shows the overall scheduling pipeline. We consider an inference server that continuously receives requests, which are inserted into the waiting queue of CHT. The scheduler builds batches incrementally in a loop. It first finds the best candidate by retrieving the waiting request with the highest overlap with the current working set. Then the RL policy evaluates the current system state and decides whether to Add the request or Stop. If Add is chosen, the request is added to the scheduler batch, and all the data structures, such as the working set and the min-heap, are updated accordingly; otherwise, the batch is dispatched to the GPU for an execution step. The decode throughput measured after the GPU execution is used as a reward for the RL policy. Completed requests are removed from the active batch, and the working set is updated.

4.2 Chunked Hash Tree (CHT)

We now describe the operations of CHT (a more formal treatment is in [32]). We begin with how token sequences are mapped to a compact representation in our system.

Hash Vectors. Given a request as a token sequence, we partition it into non-overlapping chunks of size K and build a sequence of hashes, one for each chunk, where each hash is cumulative over all tokens up to that chunk. For example, in Figure 13(a), a request with tokens A-F and $K = 2$ is represented as three incremental hashes: hash(A-B), hash(A-D), and hash(A-F). Two requests will have the same hash at level l iff they share the same first lK tokens. Each request is represented as an ordered hash vector. We use an incremental xxHash-64 state, ensuring each token is processed once, yielding a compute cost of $O(T)$, independent of the number of chunk boundaries, where T is the sequence length.



(a) Insertion **(b)** Adding to Batch **(c)** Finding the Best Request **(d)** Request Completion

Figure 13. Chunked Hash Tree Operations

Insertion. Insert is invoked when a new request arrives at the scheduler. We first compute the cumulative prefix-hash vector and store it in a map from request IDs to hash vectors for faster reuse. We also maintain a reverse index that maps each hash to the set of requests containing that hash. For each hash in the hash vector, we insert the request into this reverse index. The overall cost is dominated by the hash computation phase, which runs in $O(T)$.

Working Set, Min-Heap and Shared Prefix Tip. To evaluate how well a request matches the active batch, we maintain a working set that contains (level, hash) tuples representing prefix hashes at that level covered by currently active requests. For each request, we also maintain a missing count, which is the number of levels for which this request's hashes are not present in the working set. We insert these missing counts into a min-heap, which enables efficient retrieval of the most suitable request for addition to the batch. Insert updates both of these structures. Moreover, we do not perform in-place updates and use lazy deletion to discard stale entries in the heap. To relate to our key metric, which is the number of chunks shared by all active requests, we also maintain a shared prefix tip, storing the depth and hash of the deepest node that is shared by all the active requests. Figure 13(b) shows the Chunked Hash Tree after the request R_1 is added to the batch. The nodes marked in green denote the working set. Since there is just a single request, the shared prefix tip is at Level-3 of R_1. The heap contains the missing counts of the waiting requests along with stale entries.

Finding the Best Request. The goal of FindBest is to select the waiting request that best complements the active batch, i.e., the one with the minimum missing count, so that we maximize the depth of the shared prefix tip[3]. Since the min-heap is keyed by this count, the best candidate appears at the top after skipping stale entries. As shown in Figure 13(c), R_2 appears at the top of the min-heap and is therefore selected as the best request. Note that adding R_2 to the batch reduces the shared prefix tip by one level. FindBest runs in $O(\log W)$, where W is the number of waiting requests. For the selected request, we compute two metrics that are inputs to the RL policy (§4.3), which needs to decide whether adding the request to the current batch is beneficial for the overall throughput. One of the metrics is the change in the depth of the shared prefix tip, and the other is the number of additional waiting requests that are present at this new depth, which can all be added to the batch without any further impact on the shared prefix tip in the future.

Adding a Request to the Active Batch. When a request is promoted to the active batch, we update the working set to include its prefixes and recompute the shared prefix tip. We iterate over the hash vector backwards; for each level, if the corresponding prefix hash isn't present in the working set, it is inserted, and all waiting requests sharing it have their missing count decremented. For the shared prefix tip, admitting a new request can only *shorten* the common prefix, so we seek the deepest level where it agrees with the existing tip hash. This corresponds to the least common ancestor of all active request leaves in the hash trie. AddToBatch runs in $O(C \cdot W_h \cdot \log W)$ time, where $C = \lceil T/K \rceil$ is the number of chunks for a request with T tokens, and W_h is the number of waiting requests that share the same hash ($W_h \ll W$).

Removing a Request from the Active Batch. Finish is called when a request completes, and we remove it from the active batch. It is the symmetric counterpart of AddToBatch. We iterate over the hash vector from the start. A hash unused by any active request is removed from the working set, and all waiting requests containing it have their missing count incremented. A request completion can only *lengthen* the shared prefix tip since the remaining active requests may agree on a longer common prefix. Starting from the tip's previous position, we scan forward, checking whether all remaining active requests share the same hash at each level. (We do this efficiently by maintaining reference counts of each hash.) The extension stops at the first level of disagreement or at the shortest active request. This scan is inexpensive in practice, as the tip rarely advances more than a few levels. Finish also runs in $O(C \cdot W_h \cdot \log W)$ time. Figure 13(d)

[3]An astute reader might ask why we are choosing a request based on its difference from the working set rather than its impact on the length of the shared prefix tip. A brief explanation is that we want to pick a request that helps us revert back to a longer shared prefix tip sooner, and a request that differs from the working set by the least number of chunks helps us achieve this goal best. We refer the reader to [32] for more details.

shows the state after R_1 completes: the tip extends to R_2's final level, and waiting heap costs are updated accordingly.

Optimizations. We employ several optimizations to improve efficiency. First, FindBest may be invoked multiple times within a scheduling round without state changes, so we cache its last result and reuse it until it is invalidated by any mutation (e.g., AddToBatch and Finish). Second, we use lazy heap updates: when a request's missing count changes, we push a new entry instead of updating it in place and skip stale entries during FindBest. Third, we maintain the shared prefix tip incrementally rather than recomputing it. This drops the cost from $O(A{\cdot}C)$, where A is the number of active requests, and C is the number of chunks, to just $O(C)$ in the worst case and typically terminates earlier. Finally, all major data structures are pre-allocated at initialization to eliminate rehashing and reallocation overhead.

4.3 Batching and Reinforcement Learning

The CHT efficiently identifies the next best request to add, maximizing shared prefix length, but it does not determine *when to stop*. While adding requests improves GPU utilization, it can also shrink the shared prefix and degrade locality. We frame this stop/continue decision as a learning problem and present three policies: a heuristic, a contextual bandit, and a Q-learning agent. The scheduler incrementally builds a batch by selecting candidates via the CHT and deciding whether to admit them or dispatch the batch.

State, Actions, and Reward. At each step of batch formation, the scheduler observes the state $s = (b,\ \Delta,\ w)$ and takes an action $a \in \{\text{Add}, \text{Stop}\}$. Here, b is the current batch size (a proxy for GPU utilization), Δ is the loss in the shared prefix, and w is the number of waiting requests sharing the prefix up to the new depth, indicating whether the locality cost of this addition will be compensated by future activations. Both RL policies optimize the reward $\mathcal{R} = \text{throughput}_{\text{decode}}$. Maximizing this encourages large batches and short decode times (better prefix locality). The state is discretized into $(\hat{b}, \hat{\Delta}, \hat{w})$, where b and w use exponential bins, and Δ uses coarse thresholds. The resulting table is small, hashable, and converges quickly under moderate load.

We choose decode throughput as the reward because it is effectively the inverse of time between tokens (TBT), while remaining a safer objective than minimizing TBT directly — a batch of size one trivially minimizes per-token latency without regard for GPU utilization, so a raw TBT-minimizing policy would degenerate toward starving the batch. Time to first token (TTFT) is also a weaker signal for this decision, as it is dominated by prefill compute and queuing delay rather than by the decode batch composition choice the policy makes at each step. Finally, decode throughput implicitly captures hardware and model-specific effects — such as attention cost scaling and memory bandwidth limits — that are not exposed as explicit state features, allowing the same reward formulation to generalize across the configurations in §5 without redesigning the state space.

Heuristic Policy. As a baseline, we encode domain intuition as simple rules balancing two goals: maximizing GPU utilization (large batches) and preserving prefix locality (avoiding large drops in the shared chain). The heuristic always accepts candidates with zero chunk loss (free additions). When the batch is small, it tolerates higher loss to avoid underutilization. Otherwise, it admits a candidate only if the chunk loss is within a fixed threshold, with slightly relaxed tolerance when many waiting requests share the resulting prefix (anticipating amortization). If none of these conditions hold, it stops and dispatches the batch. This policy requires no training and converges immediately, but its fixed thresholds cannot adapt to workload or hardware changes.

Contextual Bandit. To avoid hand-tuned thresholds, we use a contextual bandit with Upper Confidence Bound (UCB) selection [5]. Each batch-construction step is treated as an independent decision: given the discretized state $\hat{s} = (\hat{b}, \hat{\Delta}, \hat{w})$, the policy selects the action that maximizes

$$\text{UCB}(\hat{s}, a) = \frac{\mu(\hat{s}, a)}{n(\hat{s}, a)} + c\sqrt{\frac{\ln S}{n(\hat{s}, a)}}.$$

Here, $\mu(\hat{s}, a)$ is the cumulative reward, $n(\hat{s}, a)$ is the number of times action a was taken in $\hat{s}$, S is the total number of decisions, and c is the exploration coefficient. The first term favors high-reward actions, while the second promotes exploration of under-sampled ones.

Q-Learning Policy. As an alternative to the previous policy, we can also model batch construction as an episodic reinforcement learning problem, where each episode begins when forming a new batch and ends upon Stop or when the waiting queue is exhausted. The Q-table [37] stores values $Q(s, a)$ for each (discretized state, action) pair and is updated after each episode using:

$$Q(s, a) \ \leftarrow\ Q(s, a) + \alpha\big[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big].$$

Here, r is the observed throughput, s' is the next state, α the learning rate, and γ the discount factor, which biases the agent toward near-term throughput, consistent with real-time inference. We follow an ε-greedy exploration strategy: at each step, a random action is selected with probability ε, and the greedy action $\arg\max_a Q(s, a)$ otherwise. During training, ε decays from 1.0 to $\varepsilon_{\min}$, transitioning from full exploration to near-deterministic behavior. Qualitatively, we expect this policy to perform better in the long term when actions influence future batch composition.

At each step, FindBest returns the lowest-cost candidate along with (Δ, w), forming the state s. The policy (heuristic,

bandit, or Q-learning) then selects ADD or STOP. If ADD, the request is activated and appended to the batch; otherwise, the batch is dispatched. After execution, the observed decode throughput is used as the reward: the bandit updates its statistics and the Q-learning agent applies the Bellman update. This has also been shown in Figure 12.

4.4 Integration with vLLM and SGLang

We integrate our scheduler as a lightweight layer on top of both vLLM [17] and SGLang [46]. The Chunked Hash Tree (implemented in C++) along with shared prefix chain tracking and reinforcement learning policies operates entirely at the scheduler level. Our code is available at https://github.com/networkedsystemsIITB/Feather.

5 Evaluation

We evaluate FEATHER across diverse models, GPU configurations, and real-world datasets and scenarios. Our evaluation answers the following questions: (i) Under what workloads do smaller homogeneous batches outperform larger heterogeneous ones? (ii) How effective is FEATHER's Chunked Hash Tree compared to traditional radix-tree-based approaches? (iii) How do different design components of FEATHER (RL, CHT) interact to produce end-to-end performance gains?

5.1 Experimental Setup

Models and Hardware. We evaluate three model families across multiple sizes: Qwen 0.5B, 1.5B, 8B [6], LLaMA 3 8B [2], and LongChat 13B [19]. Experiments are run on an NVIDIA RTX 6000 Ada GPU [23] with 48 GB GDDR6, 96 MB L2, and the maximum batch size is fixed at 500, which fits within the GPU memory for most settings. We additionally evaluate on an A100-80GB GPU for comparisons involving PAT [43], as it is specifically optimized for this hardware.

Workloads. Following prior work, request arrivals are modeled as a Poisson process, with length pruning applied as required by each experiment. L-Eval [3] is a benchmark of human-labeled query–response pairs spanning tasks such as summarization and question answering, with sequence lengths ranging from 2.7K to 210.5K tokens. LongBench [7] is a collection of long-context samples across six task categories: multi-document QA, summarization, and code completion, with context lengths between 4K and 10K tokens.

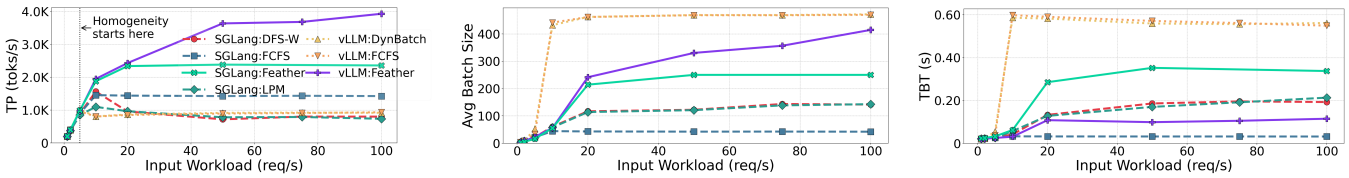
Baselines. We compare FEATHER against state-of-the-art LLM serving systems, all with prefix caching enabled. vLLM [17] uses its default FCFS scheduling policy. SGLang [46] is evaluated under FCFS, LPM, and DFS-W to cover its design space. We also evaluate DynamicBatching [28] and PAT [43], both implemented on top of vLLM: DynamicBatching adjusts batch sizes based on runtime memory utilization and latency constraints, while PAT is a prefix-aware attention kernel that reduces redundant KV cache accesses via query packing. FEATHER operates purely at the batch formation layer and is agnostic to how prefill tokens are scheduled, so it composes naturally with chunked-prefill and prefill-decode multiplexing techniques such as Sarathi-Serve [1]. Chunked prefill is enabled by default in vLLM, the backend used throughout our experiments unless otherwise noted.

Metrics. We report three key metrics. Throughput (toks/s) measures the output tokens generated per second, reflecting serving capacity and GPU utilization. Time between tokens (TBT) measures the average interval between decoded tokens, reflecting responsiveness, especially in interactive settings. We also track the average batch size per forward pass to relate scheduling decisions to these metrics.

5.2 End-to-End Comparison: FEATHER vs Baselines

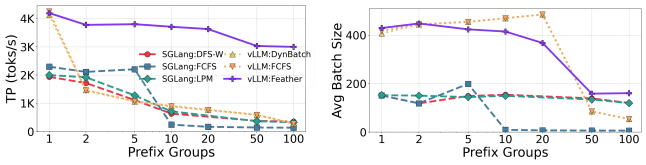
We evaluate a workload where the default configuration has 5 prefix groups, each 5K tokens long, with roughly equal request distribution, a request rate of 100 req/s, and 200 output tokens per request. Such workload configurations are common in prior work [27, 43] evaluating LLM serving systems. We vary these parameters to examine how FEATHER compares to the baselines. We also evaluate workloads without prefix sharing and show FEATHER performs on par with FCFS. The token budget is 32K tokens. For FEATHER's RL policy, we find that bandit and Q-learning approaches perform similarly across most settings; unless stated otherwise, we report results using the bandit policy.

Varying Input Request Rate. Figure 14(a) shows throughput (toks/s) as a function of the request rate. Across all policies, throughput increases with the request rate before saturating. FEATHER (on vLLM) consistently outperforms all baselines at every request rate. At 100 req/s, FEATHER achieves roughly 4× the throughput of vLLM (FCFS). Notably, this advantage is not driven by larger batch sizes. As shown in Figure 14(b), FEATHER in general maintains a *smaller* average batch size than vLLM (FCFS), and beyond the vertical line shown in the plot, FEATHER's RL policy decides to have completely homogeneous batches. This reinforces a key insight: *smaller, homogeneous batches outperform larger, heterogeneous ones*. The gradual growth in FEATHER's batch size with increasing request rate reflects its ability to efficiently aggregate more requests from the same prefix group into the active batch without sacrificing locality. SGLang policies consistently operate at smaller batch sizes than vLLM, as vLLM avoids double-counting shared prefix tokens toward the token budget, allowing more requests per batch. Dynamic Batching performs similarly to vLLM (FCFS), as all KV caches fit within GPU memory. Figure 14(c) shows time between tokens (TBT). FEATHER (on vLLM) achieves substantially lower TBT than all baselines except SGLang (FCFS), whose smaller batch sizes yield faster per-token generation. While SGLang's LPM and DFS-W policies also use small batch sizes, their TBT remains significantly higher



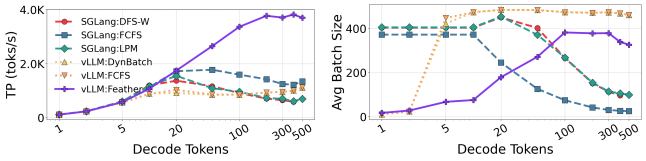
(a) Throughput **(b)** Average Batch Size **(c)** Time between tokens (TBT)

Figure 14. Poisson Workload



(a) Throughput **(b)** Average Batch Size

Figure 15. Varying Number of Prefix Groups



(a) Throughput **(b)** Average Batch Size

Figure 16. Varying Number of Decode Tokens

than Feather due to non-trivial scheduling overhead during batch construction. Feather implemented on SGLang also shows similar qualitative trends and outperforms all other SGLang policies. Unless otherwise stated, all subsequent results refer to Feather implemented on vLLM.

Varying Number of Prefix Groups. Figures 15(a) and 15(b) show throughput and average batch size as functions of the number of prefix groups. Increasing the number of prefix groups introduces greater request heterogeneity, reducing KV cache locality. This leads to a sharp throughput drop for vLLM (FCFS), even when increasing from 1 to 2 prefix groups. As the number of groups increases further, FCFS degrades rapidly: beyond 20 groups, the KV cache footprint exceeds HBM capacity, causing a sharp reduction in batch size and increased recomputation. Feather mitigates this by grouping requests with shared prefixes, increasing KV cache reuse and avoiding premature evictions. As a result, Feather maintains a more stable batch size and throughput across all prefix group configurations, significantly outperforming all baselines. At 100 prefix groups, Feather achieves 10× higher throughput than vLLM (FCFS). SGLang's LPM and DFS-W policies partially mitigate evictions, achieving higher throughput than FCFS (SGLang) at larger group counts.

Varying Number of Decode Tokens. Figures 16(a) and 16(b) show throughput and average batch size as functions of decode tokens per request. Each decode requires a full sweep over the KV cache. The policies exhibit two regimes separated by a crossover around 20 decode tokens. In the prefill-dominated regime (1–20 tokens), all policies behave similarly: throughput increases modestly, and batch sizes converge as prefills limit the impact of locality-aware scheduling. Beyond 20 tokens, the policies diverge sharply. Feather scales strongly with decode length, reaching ~3800 toks/s at 200 tokens—over 3× higher than SGLang policies. This gain arises from two factors. First, longer decode phases amplify locality across repeated KV cache sweeps. Second, a greater number of decode iterations allows the scheduler to accumulate more same-prefix requests, forming larger, more homogeneous batches. Together, these effects drive the steady increase in Feather's batch size and throughput. SGLang policies follow the opposite trajectory in batch size. Because shared prefix tokens count toward the token budget, each prefill consumes ~5K tokens, while each decode step contributes only one token. As the decode length increases, fewer prefills occur per step, reducing the number of new requests for batching.

Varying Number of Shared Prefix Tokens. Figure 17 shows throughput across the length of the shared prefix. As expected, longer prefixes increase prefill compute and KV cache pressure, reducing throughput across policies. Feather's advantage grows with prefix length, driven by higher locality gains as sequences lengthen.

Varying Models. Feather consistently achieves the highest throughput across models and outperforms baselines in nearly all cases. Its relative throughput advantage increases with model size, as larger models spend more time in attention, amplifying the benefits of KV-cache locality exploited by Feather. All graphs are shown in [32]. We also compare Feather with other policies [32] for the LongChat 13B model, where we achieve up to 22× higher throughput compared to vLLM's FCFS for larger number of prefix groups.

Varying Radix Tree Levels. Figure 18 shows throughput under different radix-tree sharing levels (seen in §3.1). All experiments use a fixed total sequence length of 4000 tokens per request. For baseline policies like vLLM's FCFS, going from 1 to 2 or more levels of the radix tree reduces the locality in KV cache traversal and, hence, degrades throughput. In contrast, Feather actively preserves prefix homogeneity within batches, leading to stable throughput at all levels.

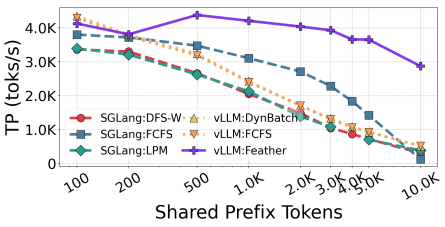


Figure 17. Varying Shared Prefix Tokens

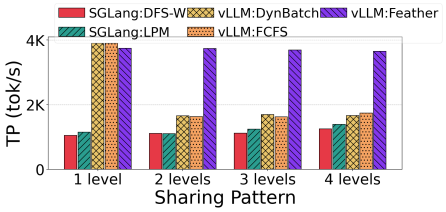


Figure 18. Varying Radix Tree Sharing Levels

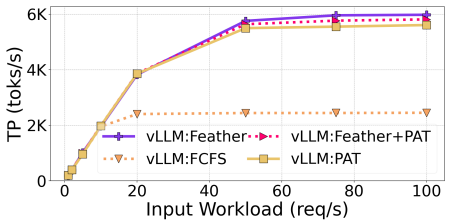


Figure 19. FEATHER vs PAT

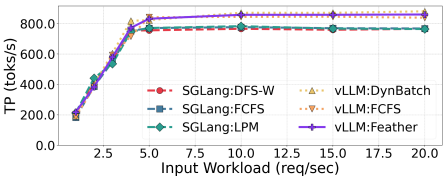


Figure 20. Throughput with no prefix sharing

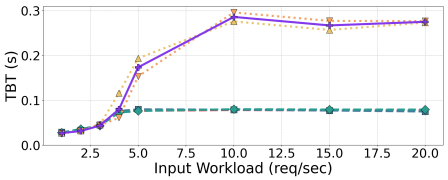


Figure 21. Time between tokens (TBT) with no prefix sharing

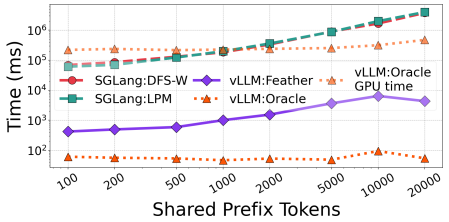


Figure 22. CPU overhead across scheduling policies

Comparison with PAT. Figure 19 compares throughput across varying request rates for PAT, vLLM (FCFS), and FEATHER on an A100-80GB GPU. PAT is specifically optimized for this hardware through custom prefix-aware kernels. Despite this, FEATHER consistently matches or outperforms PAT across all request rates, demonstrating strong performance even on different hardware. This is particularly notable because FEATHER operates purely at the scheduling layer, without any kernel-level modifications. A naive integration of FEATHER and PAT performs quite similarly to FEATHER over most regimes. As future work, we aim to more tightly integrate FEATHER with prefix-aware attention kernels, enabling deeper co-design between the scheduler and kernel-level optimizations to further improve performance.

Workloads with No Prefix Sharing. We next evaluate FEATHER and compare it against other scheduling policies on a workload without prefix sharing; that is, no two requests share a common prefix. Figures 20 and 21 report throughput and time between tokens (TBT) across a range of input rates. At higher input rates, vLLM-based policies begin to outperform SGLang policies, primarily due to their ability to form larger batches. Notably, even in the absence of prefix sharing, FEATHER, powered by its adaptive RL-based policy, matches the performance of vLLM's FCFS scheduler. In contrast, SGLang policies achieve lower TBT, which can be attributed to their smaller batch sizes.

Latency and Starvation. While our main evaluation focuses on aggregate throughput and TBT, a natural concern is whether FEATHER's prefix-homogeneous batching starves

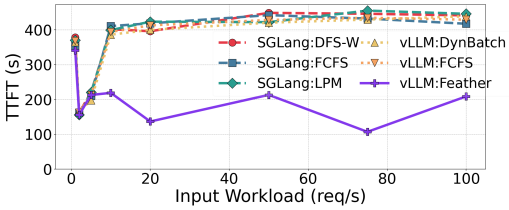
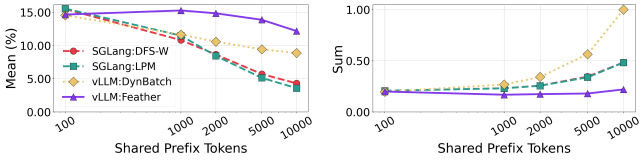


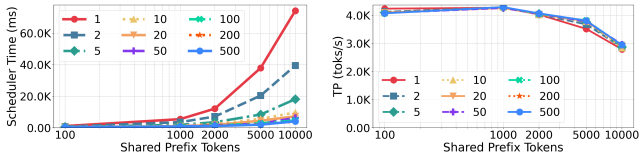
Figure 23. Time to First Token (TTFT) vs. req/sec

less popular prefix groups in favor of larger ones. Two design choices mitigate this risk. First, within FINDBEST's min-heap, entries with equal missing count are ordered by request ID, which corresponds to arrival order. Among requests that are otherwise equally compatible with the working set, earlier-arriving requests are therefore preferred, preventing indefinite deferral of any single request. Second, the exploration phase of the RL policy (§4.3) periodically constructs heterogeneous batches that include less popular prefixes, which ensures prefixes are not excluded from scheduling over the long term. As shown in Figure 23, FEATHER consistently achieves substantially lower TTFT than all SGLang and vLLM baselines across the tested workload range. This gap stems from FEATHER's prefix-homogeneous batching completing requests faster on average, which frees scheduling slots sooner and lets newly arriving requests enter execution with shorter queueing delay, rather than waiting behind a backlog of slow, heterogeneous batches. The RL policy's periodic exploration further contributes to this outcome. By regularly admitting less popular prefixes into the batch rather than deferring them indefinitely, FEATHER reduces starvation risk in the tested workloads.



(a) Mean **(b)** Sum

Figure 24. DRAM Bandwidth Utilization



(a) CPU Scheduler Overhead **(b)** Throughput

Figure 25. Sensitivity to Chunk Size

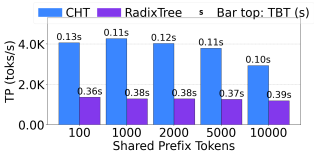


Figure 26. CHT vs. Radix Tree

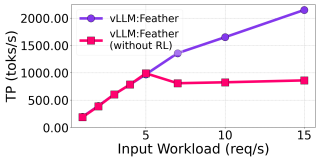


Figure 27. Feather: with and without RL

5.3 Micro-Benchmarks

Comparing Scheduler Compute Overhead across Different Policies. Figure 22 shows the CPU scheduling overhead for each policy across different shared prefix lengths. We include an Oracle policy where the caller supplies prefix group identifiers with each request. This allows the scheduler to batch same-prefix requests without any prefix matching overhead, resulting in effectively zero scheduling cost while preserving locality. We also show the GPU execution time for this Oracle scheduling policy. All three realistic policies—Feather, LPM, and DFS-W— incur higher overhead as the prefix length increases due to deeper radix-tree traversals and larger key comparisons. However, their growth rates diverge significantly. At 20K tokens, LPM and DFS-W incur roughly 1,000× higher overhead than Feather. At larger sequence lengths, their scheduling overhead exceeds the Oracle GPU execution time by up to 10×. This indicates that CPU overhead alone can negate the benefits of cache-aware batching. In contrast, Feather remains below 1% of GPU execution time across the entire range. We do a detailed analysis of the overheads of LPM and DFS-W policies in [32].

Mean DRAM Activity across Various Sequence Lengths. Figure 24(a) shows mean DRAM bandwidth utilization, while Figure 24(b) shows the total DRAM usage over the full inference duration (a proxy for total memory accessed) across policies and shared prefix lengths. Feather consistently achieves the highest bandwidth utilization due to improved spatial locality, leading to more effective prefetching. Furthermore, more homogeneous batches also increase temporal locality, which is reflected in lower total DRAM usage. SGLang's DFS-W and LPM exhibit lower mean DRAM utilization because of CPU scheduler stalls. However, they also achieve lower normalized total DRAM usage, as their prefix-aware scheduling occasionally forms homogeneous batches, allowing them to partially benefit from memory locality.

Effect of Chunk Size. Figures 25(a) and 25(b) show CPU overhead and throughput across shared prefix lengths for different chunk sizes in Feather's CHT. Increasing the chunk size reduces CPU overhead by enabling coarser-grained matching; a chunk size of 1 is equivalent to an exact radix tree. Overhead rises with prefix length due to more hash comparisons, while throughput declines due to higher KV cache memory pressure. At larger prefix lengths, the overhead gap between chunk size 1 and larger sizes widens substantially, yet throughput remains similar across all chunk sizes, suggesting that the CPU scheduling cost, even at the finest granularity, is not the primary bottleneck in our CHT. Overall, CHT is robust against chunk size selection, and even its radix tree-equivalent configuration (chunk size 1) achieves significantly better efficiency than a conventional radix tree.

5.4 Ablation Study

What if we Replace the Chunked Hash Tree with a Radix Tree? Figure 26 compares throughput and TBT when replacing CHT with a radix tree under the same RL policy. Across all prefix lengths, the radix tree yields much lower throughput. Despite homogeneous RL batching, higher CPU overhead causes frequent GPU stalls. This appears as higher TBT, indicating GPU idle time due to scheduling delays. CHT avoids this by reducing scheduling overhead, enabling higher GPU utilization and throughput.

What if We Remove RL?. Figure 27 compares throughput with and without RL at low input rates. Without RL, CHT still selects the best request, but batching never halts and always maximizes batch size. At <5 req/s, removing RL has little effect. Both fall back to FCFS, adding requests to keep GPUs utilized. Beyond this, the behaviors diverge. With RL, Feather detects homogeneous requests and stops batching at the right time, achieving higher throughput despite smaller average batch sizes. At higher rates than shown in the figure, large pools enable even non-RL to form ~500-request homogeneous batches, leading the two curves to converge. These results show that Feather's RL policy makes load-aware decisions: it falls back to FCFS under low utilization to avoid GPU starvation, and closes batch formation when homogeneity is achievable, maximizing overall throughput.

Convergence of Bandit Policy. Having shown that RL improves throughput over a non-RL baseline, we also evaluate how quickly Feather's bandit-based policy converges to

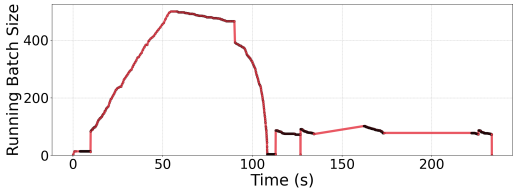


Figure 28. Convergence of Bandit Policy

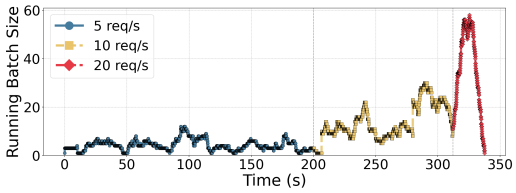


Figure 29. Performance of Bandit across Varying Workload

a stable and efficient batching configuration. We construct a workload with five prefix groups, each containing 5K tokens, and issue requests at a rate of 5 per second. Figure 28 illustrates how the running batch size evolves over time for Feather under its bandit-based policy. At the beginning, the system enters an exploration phase, during which Feather experiments with different configurations and reaches relatively large batch sizes (around 500). However, these batches are heterogeneous. After this phase concludes (around 100 seconds), the system realizes that homogeneous batching yields higher throughput, and converges to a significantly smaller batch size for the remainder of the experiment.

RL Policy across Varying Workload. In all experiments so far, including those across models [32] and hardware (Figure 19), the RL policy was reset and learned online from scratch, with no state carried over from other models or GPUs. Next, we generate a workload consisting of five prefix groups, each containing 5K tokens. The input request rate varies over time—from 5 to 10 to 20 requests per second—and we measure the average batch size as a function of time for Feather using its Bandit policy. We see in Figure 29 that the average batch size increases in response to increasing incoming load that makes larger homogeneous batches possible. Note that fluctuations in the plot are due both to RL exploration and the workload being Poisson. Because Feather's RL state and reward are defined relative to the current workload and hardware rather than in absolute terms, the learned policy generalizes across configurations without retraining.

Interested readers can see [32] for additional results.

6 Conclusion

In this paper, we showed that maximizing batch size alone is not sufficient for efficient LLM serving during the decode phase. Instead, we identified *prefix homogeneity*, the extent of the prefix shared across *all* requests in a batch, as a more important factor for improving memory locality and reducing redundant KV cache accesses. Higher batch sizes lead to better GPU utilization but also reduce the length of the prefix shared across all requests. Existing approaches ignore this aspect and also rely on expensive prefix detection mechanisms that introduce significant CPU overhead. To address these gaps, we proposed Feather, a lightweight prefix-aware scheduler that balances this trade-off between batch size and prefix homogeneity. It uses a Chunked Hash Tree (CHT) for fast prefix detection and a reinforcement learning-based batching policy that decides when to stop batch formation based on the loss in prefix homogeneity due to the addition of a new request. Our results show that Feather consistently improves end-to-end throughput across diverse workloads while keeping scheduling overhead low. As future work, we plan to extend this approach to distributed and multi-GPU settings. We are also looking into more adaptive and lightweight learning policies that can improve robustness across different workloads and deployment scenarios.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation.

[2] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[3] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2024. L-Eval: Instituting Standardized Evaluation for Long Context Language Models. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).

[4] Anthropic. 2024. Claude. https://claude.ai.

[5] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. Machine Learning (2002).

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. arXiv preprint arXiv:2309.16609 (2023).

[7] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2025. LongBench v2: Towards Deeper Understanding and Reasoning on Realistic Long-context Multitasks. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya

Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In Proceedings of the 34th International Conference on Neural Information Processing Systems.

[9] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. 2023. QuIP: 2-bit quantization of large language models with guarantees. In Proceedings of the 37th International Conference on Neural Information Processing Systems.

[10] Sumit Kumar Dam, Choong Seon Hong, Yu Qiao, and Chaoning Zhang. 2024. A Complete Survey on LLM-based AI Chatbots. https://arxiv.org/abs/2406.16937

[11] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In Proceedings of the 36th International Conference on Neural Information Processing Systems.

[12] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. https://arxiv.org/abs/2412.19437

[13] Xin Luna Dong, Seungwhan Moon, Yifan Ethan Xu, Kshitiz Malik, and Zhou Yu. 2023. Towards Next-Generation Intelligent Assistants Leveraging LLM Techniques. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.

[14] Google. 2024. Gemini. https://gemini.google.com.

[15] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. https://arxiv.org/abs/2409.12186

[16] 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. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.

[17] 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.

[18] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In Advances in Neural Information Processing Systems. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[19] Dacheng Li, Rulin Shao, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. How Long Can Open-Source LLMs Truly Promise on Context Length? https://lmsys.org/blog/2023-06-29-longchat

[20] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Guangxuan Xiao, and Song Han. 2025. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. (2025).

[21] Zejia Lin, Hongxin Xu, Guanyi Chen, Zhiguang Chen, Yutong Lu, and Xianwei Zhang. 2026. Bullet: Boosting GPU Utilization for LLM Serving via Dynamic Spatial-Temporal Orchestration. In Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.

[22] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. IEEE Transactions on Parallel and Distributed Systems (2017).

[23] NVIDIA Corporation. 2024. NVIDIA RTX 6000 Ada Generation Datasheet. https://resources.nvidia.com/en-us-briefcase-for-datasheets/proviz-print-rtx6000-1?ncid=no-ncid

[24] NVIDIA Corporation. 2026. NVIDIA Data Center GPU Manager (DCGM) Documentation. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/.

[25] OpenAI. 2024. ChatGPT. https://chatgpt.com.

[26] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo

Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. https://arxiv.org/abs/2303.08774

[27] Zaifeng Pan, Yitong Ding, Yue Guan, Zheng Wang, Zhongkai Yu, Xulong Tang, Yida Wang, and Yufei Ding. 2025. FastTree: Optimizing Attention Kernel and Runtime for Tree-Structured LLM Inference. In Proceedings of Machine Learning and Systems. https://proceedings.mlsys.org/paper_files/paper/2025/file/96894468eb44631a32d7ebd56f9892c7-Paper-Conference.pdf

[28] Bowen Pang, Kai Li, and Feifan Wang. 2025. Optimizing LLM Inference Throughput via Memory-aware and SLA-constrained Dynamic Batching. arXiv preprint arXiv:2503.05248 (2025).

[29] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. In 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

[30] Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True Few-Shot Learning with Language Models. In Advances in Neural Information Processing Systems. https://proceedings.neurips.cc/paper_files/paper/2021/file/5c04925674920eb58467fb52ce4ef728-Paper.pdf

[31] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1.

[32] Saksham Rathi, Preeti, and Mythili Vutukuru. 2026. Requests of a Feather Must Flock Together: Batch Size vs. Prefix Homogeneity in LLM Inference. https://arxiv.org/abs/2605.06046

[33] Pol G. Recasens, Ferran Agullo, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Jordi Torres, and Josep Ll. Berral. 2025. Mind the Memory Gap: Unveiling GPU Bottlenecks in Large-Batch LLM Inference. https://arxiv.org/abs/2503.08311

[34] Qidong Su, Wei Zhao, Xin Li, Muralidhar Andoorveedu, Chenhao Jiang, Zhanda Zhu, Kevin Song, Christina Giannoula, and Gennady Pekhimenko. 2025. Seesaw: High-throughput LLM Inference via Model Re-sharding. https://arxiv.org/abs/2503.06433

[35] The Big Prompt Library Contributors. 2024. The Big Prompt Library: A Collection of Prompts, System Prompts and LLM Instructions. https://github.com/0xeb/TheBigPromptLibrary.

[36] vLLM Project Contributors. 2024. Automatic Prefix Caching. https://docs.vllm.ai/en/stable/design/prefix_caching/.

[37] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-Learning. Machine Learning (1992).

[38] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. Transactions on Machine Learning Research (2022). Survey Certification. https://openreview.net/forum?id=yzkSU5zdwD

[39] 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.

[40] Jinwei Yao, Kaiqi Chen, Kexun Zhang, Jiaxuan You, Binhang Yuan, Zeke Wang, and Tao Lin. 2024. DeFT: Decoding with Flash Tree-attention for Efficient Tree-structured LLM Inference. In International Conference on Learning Representations. https://api.semanticscholar.org/CorpusID:268819748

[41] 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 Twentieth European Conference on Computer Systems.

[42] 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).

[43] Jinjun Yi, Zhixin Zhao, Yitao Hu, Ke Yan, Weiwei Sun, Hao Wang, Laiping Zhao, Yuhao Zhang, Wenxin Li, and Keqiu Li. 2026. PAT: Accelerating LLM Decoding via Prefix-Aware Attention with Resource Efficient Multi-Tile Kernel. In Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.

[44] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, Jidong Zhai, Joseph Gonzalez, and Ion Stoica. 2025. Jenga: Effective Memory Management for Serving LLM with Heterogeneity. In Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles.

[45] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yifan Qiao, Yang Zhou, Jiarong Xing, and Ion Stoica. 2026. BlendServe: Optimizing Offline Inference with Resource-Aware Batching. In Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.

[46] 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. 2024. SGLang: efficient execution of structured language model programs.

In Proceedings of the 38th International Conference on Neural Information Processing Systems.

[47] Wanyi Zheng, Minxian Xu, Shengye Song, and Kejiang Ye. 2025. BucketServe: Bucket-Based Dynamic Batching for Smart and Efficient LLM Inference Serving. https://arxiv.org/abs/2507.17120

[48] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2025. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. https://arxiv.org/abs/2412.03594

[49] 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 Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation.