PKAS: Predictive KVCache-Aware Scheduling for Faster LLM and Transformer Inferences

Jie Ye*
Illinois Institute of Technology
Chicago, Illinois, USA
jye20@hawk.illinoistech.edu

Avinash Maurya†
Argonne National Laboratory
Lemont, USA
amaurya@anl.gov

Krishna Teja Chitty-Venkata‡
Red Hat, Inc.
Boston, USA
kchittyv@redhat.com

Bogdan Nicolae†
Argonne National Laboratory
Lemont, USA
bnicolae@anl.gov

Anthony Kougkas*
Illinois Institute of Technology
Chicago, Illinois, USA
akougkas@illinoistech.edu

Xian-He Sun*
Illinois Institute of Technology
Chicago, Illinois, USA
sun@illinoistech.edu

Abstract

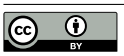
With rising popularity of LLMs, the performance, scalability, and resource-efficiency of inferences become a crucial challenge. The core part of the inference process is the KV cache, which avoids recomputing intermediate attention states, and the batching strategy that batches multiple requests per forward pass to leverage GPU parallelism. KV cache memory grows linearly with sequence length and batch sizes, easily exceeding the limited GPU memory capacity. State-of-the-art inference runtimes use continuous batching to maximize GPU utilization by interleaving the processing of new requests (i.e., prefill requests) with ongoing generation requests (i.e., decode requests). However, existing schedulers greedily admit prefill requests without considering the future KV cache memory required to successfully run the decode phases. This shortsighted approach causes frequent KV cache overflows, which in turn trigger preemption and recomputation of requests, severely degrading both throughput and latency. We propose PKAS, a Predictive KV Cache-Aware Scheduling algorithm to mitigate this inefficiency by reducing preemptions. PKAS uses a low-overhead technique to simulate future KV cache utilization and guide the admissibility for new request candidates. Combined with lightweight output-length predictions, PKAS can make better batching decisions, preventing KV cache overflows and drastically reducing preemptions. Evaluations on diverse models and workloads show that PKAS achieves up to 7.34x higher throughput and 8x lower latency compared to state-of-the-art scheduling, with the largest gains on long-context workloads where KV cache pressure is high.

CCS Concepts

• **Software and its engineering → Scheduling**; • **Computing methodologies → Neural networks**.

Keywords

KV Cache Management, LLM Inference Serving, Scheduling



HPDC '26, Cleveland, OH, USA

ACM ISBN 979-8-4007-2640-8/26/07
https://doi.org/10.1145/3806645.3807819

ACM Reference Format:
Jie Ye, Avinash Maurya, Krishna Teja Chitty-Venkata, Bogdan Nicolae, Anthony Kougkas, and Xian-He Sun. 2026. PKAS: Predictive KVCache-Aware Scheduling for Faster LLM and Transformer Inferences. In *The 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '26), July 13–16, 2026, Cleveland, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3806645.3807819

1 Introduction

Large Language Models (LLMs) have become an integral part of a wide range of industrial and scientific applications, thanks to their remarkable flexibility in tasks from text summarization and literature search [12] to complex reasoning [6]. LLMs only mark the initial step toward accelerating scientific discovery: automated data analysis, formulating hypotheses [22], and designing experiments [5]. Recently, more general transformer-based foundational models (FMs) have begun to combine multi-modal data, use domain-specific information, leverage advanced attention mechanisms (e.g., self- and cross-attention), and employ specialized architectures (e.g., mixture of experts [17]) [21]. The potential for innovations in this space has barely been tapped.

LLM/FM training and fine-tuning pose a significant challenge by requiring tens of thousands of GPUs for months at a time. Until recently, training workloads dominated HPC data centers. However, inference workloads have become the dominant nowadays. With an exploding number of LLM users, serving inference requests both efficiently and at scale is emerging as an even bigger challenge. For example, at Meta, inferences are the largest AI workload running on their HPC data centers, accounting for 65% of the energy consumption, compared to 35% consumption for pre-training [7].

The difficulty stems from the two-phase nature of LLM inferences: prefill and decode. During the prefill phase, the input prompt is processed in parallel, which is compute-intensive and effectively utilizes the GPU's processing power. The real bottleneck, however, lies in the decode phase, where the model generates one token at a time in an auto-regressive fashion. To avoid recomputation and accelerate decoding speed, a KV Cache [26] is often employed to store the Key and Value states of previously computed tokens. It typically resides in the spare GPU memory not occupied by the model parameters. This sequential, token-by-token generation is inherently memory-bandwidth bound, leaving the powerful compute units of the GPU largely idle to wait for data. To counteract this

inefficiency and maximize hardware utilization, multiple inference requests must be batched together. This, in turn, introduces significant complexity in managing the dynamic and heterogeneous states of numerous requests simultaneously.

Motivation. The KV cache capacity is usually the main bottleneck that limits the number of inference requests that can be batched together. This creates a challenging problem that demands simultaneous consideration of both scheduling and GPU memory management. Specifically, if too many requests are batched together and processed in parallel without any of them finishing, then the KV cache capacity limit will be reached prematurely, prompting the need to preempt partial requests to free up enough tokens from the KV cache for the remaining ongoing requests to continue making progress. Such preemptions will trigger recomputations later that might reduce the overall inference throughput and increase the latency of each inference request. On the other hand, if not enough requests are batched together, the GPUs and KV cache utilization remain under-utilized, which also leads to reduced throughput and increased latency. This trade-off is further complicated by the need to balance token generation throughput (maximized when using fewer, larger forward passes) with latency considerations (maximized when using more, shorter forward passes), affecting the KV cache usage patterns.

Limitations of state-of-art approaches. To address this trade-off, state-of-the-art inference runtimes such as vLLM [31], NVIDIA TensorRT-LLM [25], and HuggingFace TGI [9] typically adopt two optimizations: (1) *continuous batching*, which makes scheduling decisions at a fine granularity between individual forward passes, thus enabling new (or a previously preempted) requests to be added to an ongoing batch immediately when sufficient KV cache is available (from finished requests or spare capacity); (2) *chunked-prefill*, which allows the prefill phase of a long request to be broken into smaller chunks that are computed over multiple forward passes, thereby reducing the inter-token latency and increasing the frequency of checking for finished requests and reclaiming their KV cache tokens. Building on these optimizations, these inference runtimes usually use an optimistic greedy scheduling strategy that attempts to squeeze as many inference requests together in the same batch by checking only if there is enough KV cache capacity to run the next forward pass, without reserving capacity for future tokens. This approach assumes that previous inference requests finish often, thereby freeing up more KV cache tokens as the batch progresses. However, inference requests are increasingly getting longer, both in terms of the number of prefill tokens and decode tokens. Thus, such an optimistic strategy becomes increasingly short-sighted and might trigger a large number of preemptions, which negatively impact the performance and scalability of the inferences, both collectively and individually.

Key insights and contributions. In this work, we present *PKAS*, a Predictive and KV Cache-Aware Scheduling algorithm that aims to mitigate the aforementioned limitations via proactive KV cache simulation. In contrast to state-of-the-art approaches that make greedy, single-step admission decisions based solely on available KV cache for the next forward pass, we adopt a long-term predictive approach, which can simulate KV cache utilization over multiple future forward passes to check for potential preemptions before they occur. The simulation results are then used to guide the scheduler to decide how many new (or preempted) requests to add to the batch at each forward pass. Our KV cache simulator uses a simple and lightweight predictor to estimate the decode lengths for both ongoing requests in the batch and the candidate one, which are derived from runtime statistics of finished inference requests. Note that coarse predictions are sufficient in *PKAS* since the goal is not precise forecasting but providing the simulator with enough foresight to make informed admission decisions that can balance GPU resource utilization and inference performance. Thanks to this approach, *PKAS* reduces the number of preemptions to negligible levels while maintaining proper GPU utilizations, thereby improving both inference performance and scalability. We summarize the key contributions as follows:

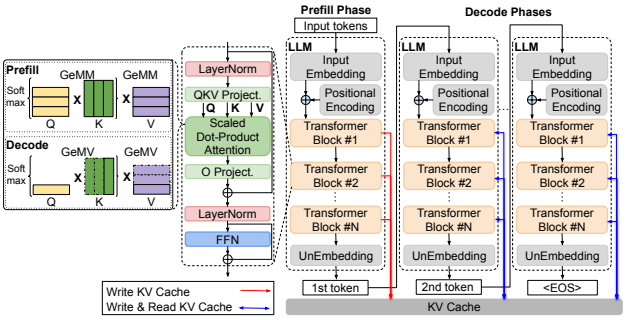


Figure 1: LLM architecture overview (decoder-only)

(1) We perform an analysis of the preemption pattern of inference requests over time to investigate the impact of recomputations on performance and identify the circumstances under which state-of-the-art scheduling strategies underperform (§ 3).
(2) Based on the gap analysis, we propose a series of design principles: isolation of admissibility criteria for new/preempted requests; admissibility recommendations based on simulated KV Cache prediction; low-overhead fast-forward simulation; and prediction-method-agnostic interface (§ 4.1).
(3) We introduce an algorithmic description of the design principles (§ 4.3) and *PKAS*, an implementation that integrates with state-of-the-art inference engine, vLLM. *PKAS* will be released as an open-source artifact upon acceptance.(§ 4.4).
(4) We evaluate *PKAS* on two models (dense and MoE) and a diverse set of real-world inference traces across various chunk sizes. The models and traces are widely used in the AI community [34]. Our approach achieves up to 7.34x higher throughput and 8x lower latency compared with the leading state-of-the-art batch strategy in vLLM (§ 5).

Limitations of the proposed approach. We assume scarce GPU memory is available for KV caching, which makes preemptions likely. Under ample KV cache capacity (unlikely in practice due to growing models and inference request sizes), our approach will not deliver an improvement of token throughput and time-per-output-token (TPOT) for batched inferences. More specifically, *PKAS* is effective for long-context workloads where KV cache pressure is high; for short-context workloads where preemptions are inherently rare, it yields modest improvements without degrading performance.

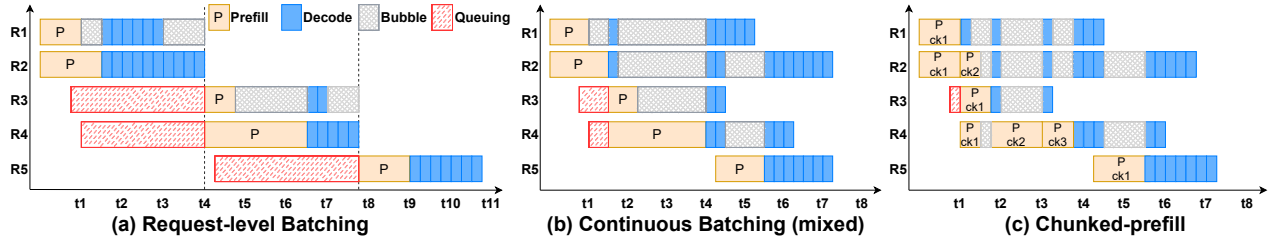


Figure 2: Illustrative comparison of different batching strategies.

2 Background

LLM Architecture. LLMs are typically based on a decoder-only transformer architecture [8, 13, 21, 30, 36] (Figure 1) that is composed of successive blocks. This part remains largely unchanged when encoders are used, which is why we can focus on decoder-only architectures without losing generality. The inference process is autoregressive in nature: after a prefill phase, in which all tokens are handled in parallel, the output is generated one token at a time during the decode phase, which appends the last output token to the input and runs a new iteration until reaching a special termination token (or a maximum number of tokens). Each iteration is essentially a forward pass in which the tokens are processed successively by each transformer block, both during prefill and decode.

KV Caching. Each transformer block uses a set of linear layers to generate Query (Q), Key (K), Value (V) vectors, and a special attention layer (often with *multiple heads* [30]) that utilizes these vectors to capture the correlations between tokens. Specifically, for input tokens $X = [x_1, x_2, ...x_n]$, three matrices $Q = X \cdot W_q$, $K = X \cdot W_k$ and $V = X \cdot W_c$ are computed using learned parameters to project each token into d_k features. Then the attention score is computed as $Attention(Q, K, V) = softmax((Q \cdot K^T)/\sqrt{d_k}) \cdot V$. Due to the autoregressive nature, a large part of the intermediate K and V results remain unchanged during the decode phase. As a consequence, K and V are cached rather than recomputed using the spare GPU memory that does not hold the LLM parameters. The KV cache is a core component of modern LLM inference runtimes [9, 14, 21, 25], making it possible to achieve a high token generation throughput. However, it is non-trivial to make efficient use of it, because the KV cache utilization grows linearly with the length of the inference request (generated during both prefill and decode), while the length keeps increasing.

Batching. Despite the KV cache's ability to reuse intermediate attention results, the sequential nature of token-by-token decoding fails to efficiently exploit the inherent parallelism of GPUs, thereby restricting token generation throughput. Consequently, optimally batching multiple requests into a forward pass has become essential for modern LLM inference runtimes. Early batching strategies, as exemplified by FasterTransformer [23], operate at coarse granularity: they group full inference requests into a new batch, run all forward passes needed to process the batch, then repeat. This process is described in Figure 2(a). A more advanced strategy, *continuous batching* [1, 25, 39], has emerged that uses finer-grained scheduling at the iteration level to further boost efficiency. In this case, the inference runtime takes scheduling decisions before each forward pass, potentially adding a new request to the batch as soon as another request has finished. This process is illustrated Figure 2(b). Given recent trends to augment inference prompts with RAG [18], splitting the prefill phase into smaller chunks is also important for reducing the duration of long forward passes, which reduces latency and increases continuous batching frequency (and thereby overall efficiency). This method, known as *chunked prefill* [14, 31, 40, 42], is shown in Figure 2(c).

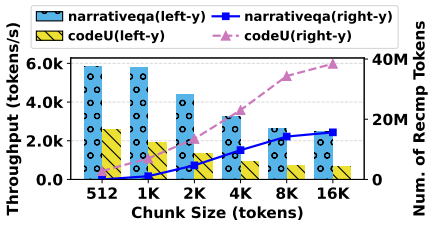
Preemption. Combining batching strategies with KV caching is subject to a difficult trade-off. As mentioned in § 1, the more inference requests are batched together, the more KV cache space is required to progress with all of them in parallel. However, the KV cache capacity is a scarce resource, since it uses the remaining GPU memory that is not allocated to model parameters. Therefore, batching too many requests together risks exhausting the KV cache space, in which case one or more requests need to be preempted to make room for the surviving requests to continue. There are two alternative strategies to resume preempted requests: swapping their KV cache tokens to host memory on preemption or recomputing the request from scratch. Recent work [37] shows that recomputation generally outperforms swapping, as the latter is constrained by slow GPU-to-Host data transfers caused by limited PCIe bandwidth.

3 Gap Analysis

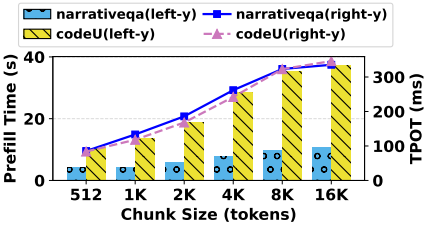
From the user perspective, it is important to minimize: (1) the time to first output token (prefill time); (2) the delay between successive output tokens (TPOT: time per output token). From the system-level perspective, it is important to maximize the inference throughput (tokens per second) in order to serve as many users as possible as quickly as possible. These objectives are often subject to a trade-off. Therefore, state-of-the-art batching strategies fill the context window of the LLM in each forward pass up to a fixed number of tokens (referred to as chunk size), which is often smaller than the model's maximum context window. The chunk size affects how many requests can be batched and how much progress each request can make in the next forward pass, which depends on whether the request is in the prefill or decode phase.

Preemptions cause significant degradation of inference performance from both the user perspective and system-level perspective. When preempted requests rejoin the batch, their recomputation inflates forward pass duration, which slows down all other inference requests, regardless whether they are in the prefill phase (i.e., longer prefill time) or decode phase (i.e., longer TPOT). Thus, regardless of how the trade-off needs to be adjusted for the batching strategy, avoiding preemptions is critical.

Intuitively, larger chunk sizes should increase the total token throughput at the expense of latency, and vice-versa. This has indeed been reported before [11]. However, preemptions lead to severe degradation of total token throughput for larger chunk sizes,



(a) Throughput vs. recompute



(b) Prefill time vs. TPOT

Figure 3: Performance analysis when running Llama3.1-8B with vLLM for different datasets and chunk sizes.

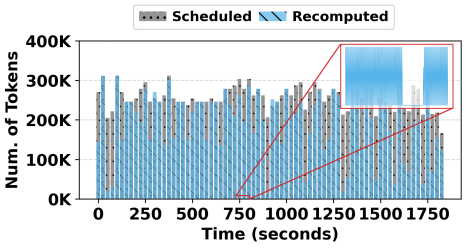


Figure 4: Num. of scheduled/recomputed tokens when using Llama3.1-8B and NarrativeQA with 8K chunk size.

especially when long inference requests are involved, thereby negating the expected throughput gains. We illustrate this in Figure 3 using *Llama3.1-8B* on two datasets with long requests (NarrativeQA and codeU). The details of the experimental setup can be found in § 5. As can be observed, in both cases, total token throughput dramatically drops for an increasing chunk size, which is correlated with the rising number of recomputed tokens. At the same time, prefill time and TPOT also increase.

To explain this phenomenon better, Figure 4 zooms into the evolution of recomputed tokens over time for NarrativeQA with an 8K chunk size. Each bar aggregates multiple forward passes over a 25s time interval. In addition to the general observation that a large number of tokens is recomputed throughout the entire duration of the inference process, there is an interesting and striking "*ping-pong*" pattern emerging, which is highlighted in the zoomed region (showing finer-grained recomputations per forward pass). Specifically, the number of recomputed tokens oscillates between high and low across consecutive forward passes (i.e., high, then low, then high again and so on). This can be attributed to a fundamental limitation of the state-of-the-art greedy scheduling strategies. They optimistically squeeze as many inference requests together in the same batch as long as there is sufficient KV cache capacity to run the next forward pass, inevitably leading to an inefficient cycle where the same request is scheduled and preempted repeatedly until another request completes and frees enough KV cache to break the cycle.

4 *PKAS* Design

Based on the gap analysis in § 3, we capitalize on the ping-pong pattern to design and develop **a novel scheduling approach** to intelligently decide *when* to admit a new candidate inference requests into an ongoing batch.

4.1 Design Principles

4.1.1 ***Isolation of Admissibility Criteria for New/Preempted Requests.*** As batching strategies and KV cache management techniques continue to evolve, it is important to address preemptions using a generic framework that is decoupled from both. In the worst case scenario, new candidates (which can be either requests that were added by users in the wait queue and never started before, or requests that were previously preempted) can be proposed by the scheduling strategy at each forward pass, which is the case of continuous batching (as discussed in § 2). Regardless of the criteria used to select new candidates to be added to an ongoing batch (e.g., priority-based or first-in first-out), the key to avoiding a preemption in the future is to admit a new candidate only if its contribution to the KV cache is unlikely to cause an overflow. Unlike state-of-the-art that applies a short-sighted admissibility check on the availability of enough KV cache tokens to run the next forward pass, we propose a far-sighted approach based on simulating future KV cache utilization. A key idea we propose is to collect and track runtime statistics independently to keep predictions self-contained and avoid side-effects such as proactive KV cache allocation or manipulation of priorities and request order. By achieving this goal, we effectively enable isolation, since the only impact on the batching strategy is the admissibility decision. We will discuss next how to achieve this goal.

4.1.2 ***Admissibility Recommendations Using Simulated KV Cache Utilization.*** Predicting whether a new request will cause a KV cache overflow is complex and cannot be solved analytically. The difficulty stems from several non-trivial factors: the current state of the KV cache, the phase (prefill/decode) and progress of each running request, and future completion events that will release cache blocks. The KV cache space is managed in blocks that are assigned to different inference requests [31]. When using prefix caching, the problem is further complicated by the need to keep track of reference counting. Thus, simply tracking the number of free tokens in the KV cache is insufficient; we must model the allocation and utilization of individual blocks without assuming a specific cache policy. Furthermore, predicting when requests will finish (releasing blocks) requires a statistical approach since termination is uncertain. Due to these complexities, we solve the problem via simulation. We introduce a *shadow* KV cache allocator that can instantly fork the current cache state and mimic the real allocator by performing fake (de)allocations that only update the block table, not the GPU memory. The simulation tracks the progress of all requests, including the new candidate, modeling when they allocate new blocks and when they release blocks upon completion. We run the simulation only for the duration required to complete the new candidate request. If no overflow occurs during this period, the request is admitted; otherwise, it is denied.

4.1.3 ***Low-Overhead Fast-Forward Simulation.*** An ongoing batch constantly changes as new or preempted requests are added and others finish after each forward pass. Because our simulation is a self-contained recommendation, the variables it depends on (state of KV cache, state of requests) may quickly diverge from

reality. Therefore, we must run the simulation frequently, ideally at every forward pass, in order to compensate for these differences. A key challenge is minimizing the simulation's overhead, as it could become a bottleneck when processing numerous requests over many forward passes. Under such circumstances, it is infeasible to simulate each individual forward pass. Instead, we propose a "fast-forward" approach to simulate multiple passes in a single step. The key insight is that we only need to track the next completion event (the number of K passes until any request finishes). Until that point, no KV cache blocks are released. This allows us to use the shadow KV cache allocator to simulate new block allocations as needed for the tokens generated by all ongoing requests over the next K steps. Then, we repeat this process after each completion event. Since all ongoing requests are known at the start of a simulation, we simply sort them by their estimated remaining tokens to determine the sequence of completion events. This fast-forward method is especially effective for long requests, allowing many passes to be skipped between actual state updates.

4.1.4 ***Prediction-Agnostic Design.*** Simulating future KV cache utilization requires predicting output lengths of inference requests. Since the same prompt can yield different output lengths across LLM models, this is inherently challenging. State-of-the-art approaches range from lightweight heuristics to trained models [41] and analysis of internal states [28]. Predictions involve a fundamental trade-off: overly optimistic estimates cause frequent preemptions, while pessimistic ones underutilize the context window. Prioritizing speed and admissibility isolation, we adopt a lightweight heuristic (detailed in § 4.3.3): we track the moving average and standard deviation of finished-request output lengths via Welford's algorithm, then bias the prediction one standard deviation below the mean. This accounts for co-batched requests finishing in the meantime and freeing KV cache space during the decode phase. The standard deviation serves as a proxy for prediction uncertainty: higher uncertainty yields a more optimistic prediction, reflecting that the request is more likely to finish sooner than estimated at admission. This has two advantages: (1) it is non-intrusive and lightweight (output length is directly observable and the statistics are computationally trivial), thus avoiding scheduling delays; (2) it is robust, adapting the optimistic bias to uncertainty to mitigate the ping-pong effect without causing underutilization. Since the simulation is re-run at every forward pass with updated states and statistics (§4.1.2), prediction errors are short-lived and self-correcting, enabling online adaptation without dataset-specific tuning. *PKAS* is most effective under high KV cache pressure (long contexts, limited GPU memory, large batches) where greedy scheduling triggers frequent preemptions, but still yields modest improvements for short-context workloads. Our approach is compatible with more sophisticated predictors; their latency could further be hidden by running them proactively and asynchronously on CPUs during forward passes, though this is outside the scope of this work.

4.2 System Composition of *PKAS*

Figure 5 the integration of PKAS's proposed design principles within vLLM inference engine, while it can be extended naturally to other inference engines using similar inference workflows (e.g., TensorRT-LLM [25], SGLang [40], etc.).

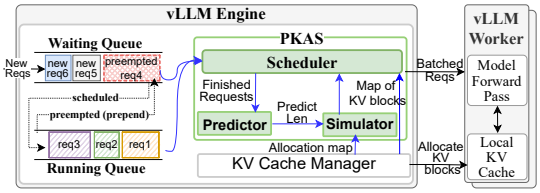


Figure 5: Architecture of *PKAS*. The green-shaded components represent PKAS's additions to the vLLM.

New inference requests submitted by the users are processed sequentially by an API server, which tokenizes raw input of each request and appends the request to a *waiting queue*. A separate *running queue* is used to track the requests that are actively generating output tokens in the ongoing batch. When requests in the running queue exhaust available KV cache blocks, at least one running request must be preempted until enough KV cache blocks are freed for the surviving requests to make progress. Preempted requests are added back to the waiting queue and are typically selected as new candidates with higher priority. The vLLM Worker executes a model forward pass for the batched requests and updates each request's key and value states in the KV cache.

PKAS employs three components (Figure 5) that collectively make admissibility decisions — determining which and how many requests from the waiting queue can be admitted to the ongoing batch to reduce preemptions in future decoding iterations. The **Scheduler** implements the core admissibility logic, building upon vLLM's hybrid continuous batching and chunked-prefill mechanism. It coordinates with the **Simulator** and **Predictor** to make informed scheduling decisions. Before admitting a new candidate (selected from the head of the waiting queue) to join the ongoing running requests, it invokes the Simulator to simulate the future KV cache utilization and then decides to admit or reject the candidate accordingly. Upon request completion, it notifies the Predictor to update the running statistics. Note that we isolated the admissibility recommendations (provided by the Simulator and Predictor) from the underlying scheduling policy (e.g., FIFO), enabling integration with other strategies. The **Simulator** performs a light-weight fast-forward simulation of future KV cache utilization (detailed in § 4.1 and § 4.3) for current active requests and the given candidate. It retrieves the block allocation map and utilization statistics for running requests from the KV Cache Manager before performing simulation, then combines this information with predicted output length from the Predictor to simulate the KV cache utilization. The **Predictor** provides the estimation of the output length using a simple deviation-aware moving average approach (detailed in § 4.3). To achieve this, it maintains aggregated statistics from completed requests, which are dynamically updated via a notification from the Scheduler. Together, these components enable *PKAS* to make smart decisions, effectively reducing preemptions without sacrificing resource utilization.

4.3 Zoom on the Scheduling and Simulation Algorithms

We use the following notations throughout: Q_w denotes the waiting queue, Q_r the running queue, b the token budget per iteration, and r a candidate request selected from Q_w. Requests in Q_w are ordered

Algorithm 1: Predictive KVCache-Aware Scheduling.

```
Input  : Q_w: waiting queue; Q_r: running queue; C_max: Max # tokens per batch; B_max: Max # requests per batch
Global : Simulator: PKAS simulator described in Algorithm 2.
Output : Batched requests to execute in current iteration
1  Function schedule():
2      R_wait ← ∅, R_run ← ∅, b ← C_max
3      R_run, b, preempt ← schedule_running(b)
4      if ¬preempt then
5          R_wait ← schedule_waiting(b)
6      return R_run ∪ R_wait
7  Function schedule_running(b):
8      B ← ∅
9      preempt ← False
10     for req ∈ Q_r do
11         // Free completed requests and update prediction statistics
12         if has_finished(req) then
13             KVmgr.free(req)
14             Predictor.update_statistics(req)
15     // Iterate Running requests in insertion order (FCFS priority)
16     for req ∈ Q_r and b > 0 do
17         // Return min(remaining prefill tokens or 1 if decoding, b)
18         τ ← cal_input_toks(req)
19         can_sched ← True
20         while ¬KVmgr.alloc(req, τ) do
               // Preempt until enough space for req
21             preempt ← True
22             victim ← Qr.pop_last()
23             KVmgr.free(victim)
24             Q_w.push_front(victim)
25             if victim == req then
26                 can_sched ← False
27                 break
28         if can_sched then
29             B ← B ∪ req, b ← b − τ
30     return B, b, preempt
31 Function schedule_waiting(b):
32     B ← ∅
33     // Iterate waiting requests in arrived order (FCFS)
34     for req ∈ Q_w and b > 0 do
35         τ ← cal_input_toks(req)
36         if ¬Simulator.fast_forward(req, τ, Q_r) then
37             break
38         KVmgr.alloc(req, τ)
39         B ← B ∪ req, b ← b − τ
40         Q_w.pop(req)
41         Q_r.push_back(req)
42     return B
```

by arrival time and processed following the First-Come-First-Served (FCFS) policy, while requests in Q_r are processed in insertion order.

4.3.1 ***Scheduling.*** The scheduling workflow for batching requests is described in Algorithm 1. Aligned with vLLM's scheduler, *PKAS* prioritizes scheduling requests already in Q_r to form a batch R_{run} (Line 3). If no running requests need to be preempted, it proceeds to admit new or resumed requests from Q_w to form batch R_{wait} (Line 5). Then the union of R_{run} and R_{wait} forms the next batch to be processed, which will be passed to the vLLM worker for performing the next forward pass.

When scheduling requests from Q_r, the scheduler proceeds as follows. The scheduler first iterates over Q_r to check if any requests have completed their generation. If a request is finished, its KV cache blocks are reclaimed and its output length is reported to the Predictor for statistics update. The completed request is then removed from Q_r. Next, for each request in Q_r, we check if it can be admitted given the remaining token budget (b) and available KV

Algorithm 2: *PKAS* Simulated KV Cache Utilization.

```
Input  : r: candidate req. to schedule; τ: number of tokens for r to process in the next forward pass; Q_r: running requests
Global : ShadowKV: shadow KV cache manager as described in § 4.1, Predictor: Predictor governed by equation 1.
Output : schedule decision for the candidate req. (boolean)
1  Function fast_forward(r, τ, Q_r):
2      // Create a shadow copy of current KV cache state
3      ShadowKV ← clone_state(KVmgr)
4      // Reject if not enough space for r to run next forward pass
5      if ¬ShadowKV.alloc(r, τ) then
6          return False
       // compute remaining Δ tokens for r and Q_r
7      // in(): prefill tokens; out(): predicted output length; progress(): tokens already processed
8      Δ_r ← in(r) + Predictor.out(r) − τ
9      for req ∈ Q_r do
10         Δ_req ← in(req) + Predictor.out(req) − progress(req)
       // sort Q_r based on ascending order of Δ
11     Q'_r ← sort(Q_r, Δ)
12     // cumulative simulated forward passes advanced so far
13     iteration ← 0
14     while |Q'_r| > 0 do
15         if Δ_r − iteration == 0 then
16             return True
17         req ← Q'_r.front()
18         // compute minimum tokens K (or forward passes) to advance until r or the closest request finishes
19         K ← min(Δ_req, Δ_r) − iteration
20         if K == 0 then
21             ShadowKV.free(req)
22             Q'_r.pop_front()
23             continue
24         // advance all surviving requests by K tokens
25         for next ∈ Q'_r do
26             if ¬ShadowKV.alloc(next, K) then
27                 return False
28         if ¬ShadowKV.alloc(r, K) then
29             return False
30         iteration ← iteration + K
31     return True
```

cache blocks. This process continues until either the token budget or the KV cache capacity is exhausted. Specifically, the request can be admitted if the KV cache manager has sufficient blocks to hold τ tokens, where τ equals the minimum of the request's demand (chunk size or the size of remaining prefill tokens if still in prefill phase, or one if in decode phase) and the remaining budget. If the allocation fails due to insufficient KV cache memory, the lowest-priority request *victim* is preempted from Q_r and retries. This process is repeated until either the allocation succeeds or the request itself is the one for preemption. Successfully admitted requests eventually form the batch R_{run}.

After all admissible running requests are processed, the scheduler starts scheduling new or previously preempted requests from Q_w. This process mirrors the same strategy used for running requests, except for one critical addition: before admitting a candidate r, it invokes the Simulator query if r is eligible for admission without causing memory exhaustion in future iterations. If the Simulator recommends admission, the candidate r is moved from Q_w to Q_r. The new admitted requests are eventually returned as R_{wait}.

4.3.2 ***Simulation.*** Algorithm 2 describes the Simulator's fast-forward simulation of future KV cache utilization. First, given the current state of the KV cache manager, we construct a shadow

copy of the KV cache (*ShadowKV*), which mirrors the KV blocks map without affecting the actual allocation state. Then, the Simulator checks if *ShadowKV* can allocate τ tokens (requested by the Scheduler) for candidate r in the next forward pass. If this initial allocation fails, r is immediately rejected. Otherwise, we query the Predictor for the predicted output length and compute the remaining number of tokens for r to be processed (denoted Δ_r). Similarly, we estimate the remaining number of tokens for each request in Q_r (denoted Δ_{req}). With these initial checks and computations in place, the simulation proceeds to the fast-forward stage.

As mentioned in § 4.1, we employ an event-driven strategy triggered by completion events to fast-forward through the forward passes. To this end, we sort the running requests in ascending order based on Δ_{req}. Then, while there are remaining running requests and r did not finish, we compute the minimum number of tokens K needed to finish either r or the closest request req with the smallest Δ_{req}. If the simulation shows a running request finishes ($K = 0$), then its blocks are freed from *ShadowKV*. In any case, we try to advance all remaining running requests and candidate r by K tokens, which may be subject to new block allocations. The allocations are performed by *ShadowKV*, leaving the actual KV cache state unchanged. If any allocation fails, the candidate r is rejected. Otherwise, it is admitted. This approach efficiently simulates the future KV cache utilization without executing actual forward passes, enabling proactive admission control.

4.3.3 ***Prediction.*** As described in § 4.1, we use a deviation-aware moving average as our Predictor to estimate the request's output length, which is used by the Simulator. To implement this, it must collect statistics about the finished requests from the Scheduler (via *Predictor.update_statistics*) and incrementally maintain a moving average and standard deviation. The predicted output length is computed using the following formula:

$$Predictor.out(req) = \begin{cases} \left\lceil \left| M1 - \sqrt{\frac{M2}{N_{f\_reqs}-1}} \right| \right\rceil, & \text{if } N_{f\_reqs} > 1 \\ 1, & \text{if } N_{f\_reqs} \leq 1 \end{cases} \quad (1)$$

where N_{f_reqs} is the number of finished requests, $M1$ is the running average output length of finished requests, and $M2$ is an auxiliary variable that tracks the running sum of the squared differences from the current mean, which is based on Welford's algorithm [33]. The intuition behind subtracting a standard deviation from the average is to leverage the reclaimed tokens from running requests that finish earlier than average, which is more beneficial than incurring occasional preemption penalties due to being over-optimistic. Of course, the prediction is adjusted by considering the progress of the request: if a request has already produced more tokens than the current estimate, then it is conservatively estimated to produce one more token in the future.

4.4 Implementation

We implement the design principles proposed in *PKAS* as an open-source modular extension to vLLM 0.8.4 in Python, focusing on enhancing the scheduler while maintaining full compatibility with vLLM's optimizations – PagedAttention, tensor parallelism, pipeline parallelism, etc. To keep changes localized, we integrate *PKAS* by extending the scheduler's scheduling policy interface (i.e., vLLM_V1's scheduler). We also provide a configuration option to support flexible switching between vLLM's scheduler and *PKAS*.

To implement the `Simulator` efficiently, we use a lightweight shadow KV manager that clones the current KV cache state with minimal overhead. Specifically, it copies the block table map and per-block utilization at the start of the simulation. During simulation, the shadow KV cache manager records two runtime statistics: (1) the blocks allocated to each request; (2) the fill rate of the last block allocated to each request. Thus, the operations for allocating tokens to a request ($ShadowKV.alloc()$) and freeing them ($ShadowKV.free()$) involve only basic arithmetic computations, making their overhead negligible. Thanks to this approach, the `Simulator` can check and advance multiple forward passes at a time with minimal cost, as described in Algorithm 2 and Algorithm 1.

Although *PKAS* is built upon vLLM, its fundamental principles are general and can be seamlessly integrated into other inference systems (e.g., SGLang and llama.cpp).

5 Evaluation

5.1 Methodology

5.1.1 ***Platform and Software.*** Our experiments are conducted on the ALCF's Polaris platform [1], a 560-node HPE Apollo 6500 Gen 10+ based system. Each node has 4×A100 GPUs with 40 GB HBM2 on each (aggregated GPU memory of 160 GB), 1× AMD EPYC Milan 7543P processor with 32 Zen3 Cores (64 threads), 1×512 GB DDR4 RAM, and 2× NVMe SSDs of 1.6 TB each. The GPUs within each node are interconnected via high-speed NVLink, providing a total bandwidth of 600 GB/s for intra-node GPU communication.

5.1.2 ***Compared Approaches.*** Throughout our evaluations, we compare the following approaches, all implemented atop vLLM [20]:

- **vLLM-FCFS (vLLM):** FCFS is vLLM's default scheduler, which admits requests aggressively based on their arrival time at each scheduling step.
- **Learning-to-Rank (Ranking)** [10]: It uses a ranking predictor to predict output lengths and rank requests by their predicted sizes before execution. The scheduler then utilizes the ranking information to guide its decisions. We employ a pre-trained opt-125m model for prediction.
- **PKAS (Ours):** PKAS applies a simulator to estimate the KV cache utilization of future steps and a moving average predictor for output length prediction. The scheduler decides when to admit new requests based on the simulation results.

Both vLLM-FCFS and PKAS implementation are based on vLLM v0.8.5, ensuring a fair comparison as our primary baseline. For Ranking, we use its public codebase built on vLLM v0.4.1 to faithfully reproduce it without without unintended discrepancies.

5.1.3 ***Models.*** We evaluate our approach on two representative models with distinct architectures: `Llama-3.1-8B` (dense model) and `Yi-Coder-9B-Chat-8x-MoE` (MoE model). The MoE model contains 8 experts per transformer block layer, with only 2 experts activated per token during inference. We selected them for their widespread adoption, different scales, and architectural diversity. Both models use `bfloat16` for model weights and KV cache,

[1] https://docs.alcf.anl.gov/polaris/hardware-overview/machine-overview/

Table 1: Model configurations. The MoE variant activates 2 of 8 experts per layer during inference.

Model	Model Size(GB)	Layers	Attn. Heads	KV Heads	Head Size	Max Avail. KV Cache Size(GB)	Max Avail. KV Cache Blocks	Per Token KV Cache Size(KB)
Llama-3.1-8B (1 GPU)	15	32	32	8	4096	16.48 GB	8440	128 KB
Yi-Coder-9B-Chat-8x-MoE (4 GPUs)	101.24	48	32	4	4096	21.48 GB	14664	96 KB

and leverage GQA (Group Query Attention) instead of traditional MHA (Multi-Head Attention) to reduce the KV cache memory footprint. Table 1 describes each model's configuration and the KV cache information, with per-token KV cache size calculated by $layers * 2 * kv\_heads * \frac{head\_size}{attn\_heads} * sizeof(data\_type)$ [37].

5.1.4 ***Datasets.*** We evaluate *PKAS* on three real-world datasets with varying sequence lengths: NarrativeQA [4], codeU [2], and ShareGPT [3], all widely adopted in prior work [34]. NarrativeQA (for reading comprehension) and codeU (for code analysis) have average input lengths of 24K and 31.5K tokens, respectively, representing the long-context workloads that are increasingly common with the expansion of modern LLM context windows. ShareGPT, a real-world dataset of human-ChatGPT conversations, features input lengths from 4 to 2.3K tokens, representing a short-context workload with high variability, which results in larger prediction deviations in our predictor (Equation 1). Since NarrativeQA and codeU have relatively few requests (200 each), we use them fully in our evaluations. Although the request count is modest, their long contexts limit the number of requests that can coexist in each forward pass, creating KV cache pressure that is representative of memory-constrained serving scenarios. ShareGPT has a much larger number of requests (50000). Thus, to reduce the duration of our experiments in order to make it feasible to experiment with a large number of combinations of parameters and compared approaches, we select a random subset of ShareGPT (5000). This subset retains the variance and long-tail distribution of the original workload.

5.1.5 ***Runtime Configurations.*** All experiments use hybrid continuous batching with chunked-prefill enabled and adopt recomputation as the preemption policy. As described in § 3, chunk size is a critical parameter that directly influences the inference performance, particularly for heavy workloads with long contexts. The chunk size corresponds to vLLM's `max_num_batched_tokens`, the primary knob for controlling batch formation (the token budget per forward pass). We vary the chunk size across a range from 512 to 16K tokens in our test. We also configure the max concurrent requests in a batch (i.e., `max_num_seqs`) to 512 for single-GPU and 1024 for multi-GPU to ensure sufficient batching pressure for both long- and short-context workloads. Multi-step scheduling is disabled to ensure a fair comparison with Ranking, which does not support it. All compared approaches use same runtime configurations. We run all experiments three times and report average results.

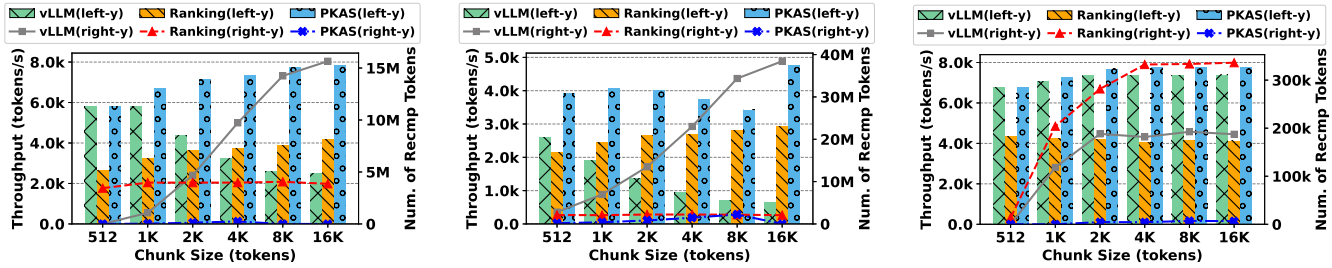
5.1.6 ***Key Metrics.*** Throughout our evaluation, we meaure: 1) throughput (i.e., the number of effective tokens processed per second, excluding recomputed tokens); 2) the total number of recomputed tokens (to quantify the computation overhead or loss of progress due to preemptions); 3) latency metrics including queuing time (the duration a request waiting in the queue before being scheduled), prefill time (the duration when a request is dequeued until it finishes prefill stage processing and gets the first output token), TTFT (i.e., Time-To-First-Token, defined as the sum of queuing and prefill time), and TPOT (i.e., Time-Per-Output-Token, referring to the average time between per output token); and 4) KV cache utilization (expressed as fraction of utilized to total KV cache).

5.2 Performance Results on a Single GPU

We first compare PKAS against other baselines in terms of inference performance when serving Llama-3.1-8B on a single GPU across three datasets. For this experiment, we configured vLLM with a maximum model context length of 32K tokens (limited by single-GPU memory) and a maximum batch size of 512 requests per forward pass to prevent KV cache underutilization when serving the short-context workloads like ShareGPT.

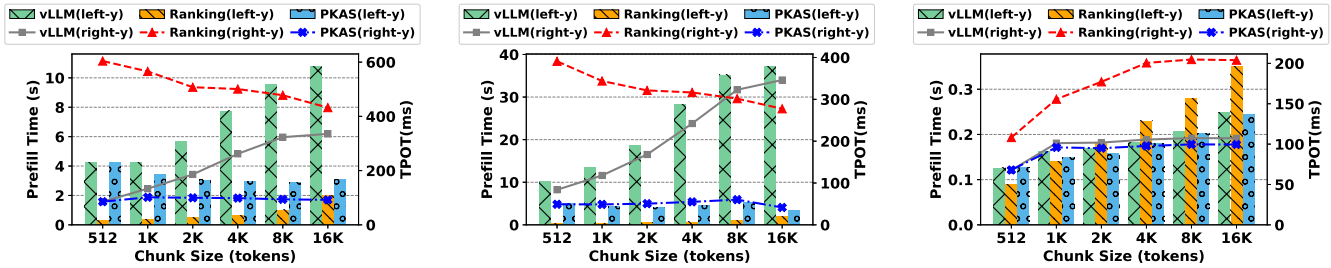
Throughput and Total Recomputed Tokens. Figure 6 illustrates the inference throughput and total recomputed tokens across the three datasets. Overall, *PKAS* (blue bars) consistently achieves higher throughput than vLLM (green bars) and Ranking (orange bars) across all chunk sizes, with the most pronounced gains on long-context datasets. On NarrativeQA (Figure 6a) and codeU (Figure 6b), *PKAS* outperforms vLLM by up to 3.12× and 7.34×, and Ranking by up to 2.19× and 1.83×, respectively. These substantial gains stem from the dramatic reduction of recomputations caused by frequent preemption. vLLM's recomputation (gray-solid line) reaches up to 15M on NarrativeQA and 40M tokens on codeU at 16K chunk size, while *PKAS* maintains near-zero recomputations across all configurations. On ShareGPT (Figure 6c), where recomputation overhead is inherently smaller due to the low per-request KV cache footprint of short contexts, *PKAS* still delivers about 5% speedup over vLLM for chunk sizes of 4K and above. For long-context datasets, vLLM's throughput decreases at larger chunk sizes because escalating recomputation overhead outweighs the batching benefits, whereas for ShareGPT, negligible recomputation allows throughput to scale with chunk size. Both *PKAS* and vLLM outperform Ranking across all chunk sizes. This performance gap may be attributed to the fact that Ranking is built upon a legacy vLLM version, lacking recent optimizations added in the current vLLM baseline. These results confirm that *PKAS*'s throughput gains directly correlate with its ability to minimize recomputation by making smart request scheduling decisions based on simulated KV cache usage and the moving average-based output length prediction.

Latency. Figure 7 presents the prefill time and the TPOT latency for the three scheduling methods across various datasets. For long-context datasets, Ranking (orange bars) achieves the lowest prefill time, with *PKAS* (blue bars) slightly higher but still substantially lower than vLLM (green bars). Ranking's lower prefill time stems from its shortest-output-first ordering, which reduces prefill contention by completing shorter requests before longer ones begin. In contrast, vLLM and *PKAS* use FCFS, where long-running decode requests compete with prefilling requests for the token budget, leading to longer prefill times. On ShareGPT, *PKAS* achieves the lowest prefill time among all three methods. Across all workloads, *PKAS* (blue-dashdot line) consistently achieves the lowest



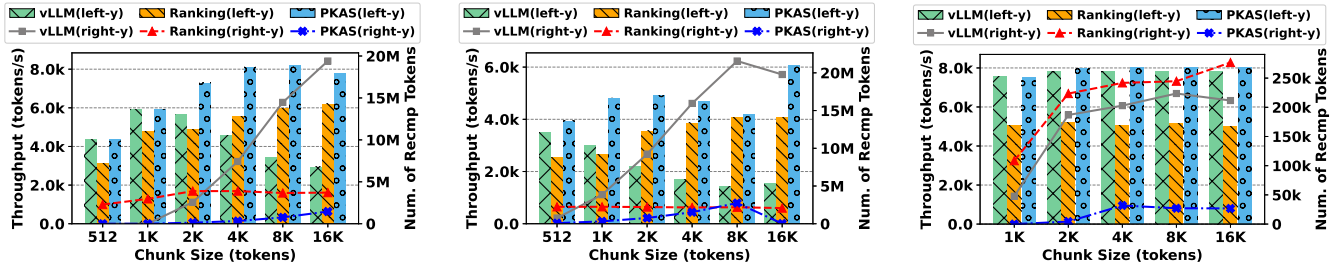
(a) NarrativeQA dataset (b) codeU dataset (c) ShareGPT dataset

Figure 6: Llama-3.1-8B (1 GPU): Throughput (higher, better) and Recomputed tokens (lower, better) for different chunk sizes.



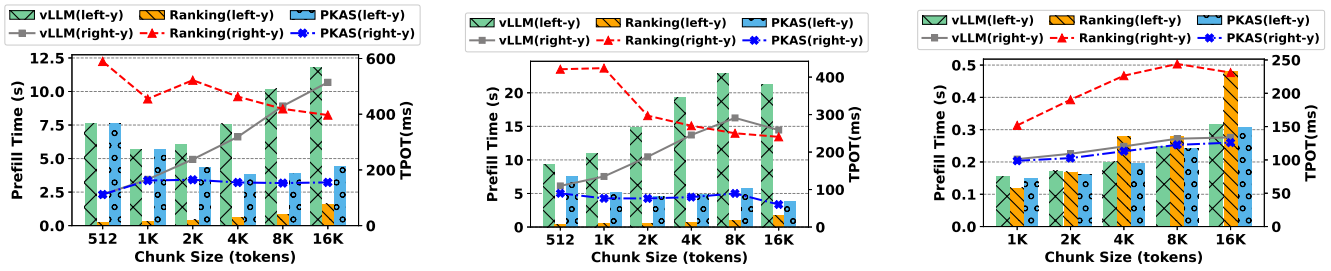
(a) NarrativeQA dataset (b) codeU dataset (c) ShareGPT dataset

Figure 7: Llama-3.1-8B (1 GPU): Prefill time (lower, better) and TPOT latency (lower, better) for different chunk sizes.



(a) NarrativeQA dataset (b) codeU dataset (c) ShareGPT dataset

Figure 8: Yi-Coder-9B-Chat-8x-MoE (4 GPUs): Throughput (higher, better) and recomputed tokens (lower, better).



(a) NarrativeQA dataset (b) codeU dataset (c) ShareGPT dataset

Figure 9: Yi-Coder-9B-Chat-8x-MoE model (4 GPUs): Prefill time (lower, better), and TPOT (lower, better).

TPOT latency, whereas Ranking (red-dashed line) incurs substantially higher TPOT, especially for smaller chunk sizes. Additionally, when using long-context datasets, *PKAS* exhibits decreasing prefill time and TPOT latency as chunk size increases because eliminating recomputations allows larger chunk size to benefit more more efficient batching, while vLLM shows the opposite trend as recomputation overhead grows with chunk size. Ranking, despite achieving moderate prefill times, suffers from substantially higher TPOT latency, which may be attributed to its dependence on a legacy vLLM. For the short-context ShareGPT, all approaches exhibit increasing TPOT as chunk size grows because each forward pass processes more tokens, increasing per-request inter-token delay. This indicates inherent workload properties rather than scheduling effects. Quantitatively, *PKAS* achieves up to 3.02× and 8.58× prefill time

Table 2: Latency comparison PKAS vs. vLLM: queuing (time waiting in queue before scheduling), prefill (time for prefill forward passes after dequeuing), and TTFT (queuing + prefill). Better approach in bold.

Llama-3.1-8B mode																				
Chunk Prefill Size →		512			1K			2K			4K			8K			16K			
Datasets	Time (s)	vLLM	Ranking	PKAS	vLLM	Ranking	PKAS	vLLM	Ranking	PKAS	vLLM	Ranking	PKAS	vLLM	Ranking	PKAS	vLLM	Ranking	PKAS
Narrative QA	Queuing	418.05	858.4	**417.95**	419.75	706.89	**363.07**	571.65	636.58	**343.28**	803.54	618.82	**338.96**	1031.64	598.62	**312.78**	930.08	563.2	**307.99**
	Prefill	4.22	**0.3**	4.22	4.27	**0.38**	3.45	5.69	**0.48**	3.05	7.75	**0.63**	2.04	9.58	**1.02**	2.9	10.76	**2.03**	3.1
	TTFT	422.27	858.7	**422.17**	424.01	707.27	**366.52**	577.34	637.06	**346.33**	811.29	619.45	**341.9**	1041.22	599.64	**315.68**	940.84	565.22	**311.1**
codeU	Queuing	458.35	448.09	**310.22**	626.6	386.33	**308.27**	893.81	353.65	**329.36**	1317.5	352.96	**375.68**	1829.38	343.27	**432.69**	2089.02	344.49	**244.56**
	Prefill	10.26	**0.38**	4.95	13.48	**0.41**	4.25	18.75	**0.49**	4.62	28.34	**0.69**	4.62	35.21	**1.03**	5.32	37.14	**2.07**	3.31
	TTFT	468.61	448.47	**315.16**	640.07	386.74	**312.52**	912.57	354.14	**333.54**	1345.84	353.65	**380.29**	1864.59	344.3	**438.01**	2126.16	346.56	**247.87**
shareGPT	Queuing	**132.79**	264.17	133.58	121.64	259.62	**118.98**	115.59	259.87	**110.94**	115.27	264.77	**109.52**	115.1	261.94	**109.97**	114.51	262.86	**109.42**
	Prefill	0.13	**0.09**	0.13	0.16	**0.14**	0.15	0.17	0.18	**0.16**	0.18	0.23	0.18	0.21	0.28	**0.2**	0.25	0.35	0.25
	TTFT	**132.92**	264.26	133.7	121.8	259.75	**119.13**	115.76	260.05	**111.1**	115.45	265.0	**109.7**	115.3	262.22	**110.18**	114.76	263.21	**109.67**
Yi-Coder-9B-Chat-8x-MoE																				
Narrative QA	Queuing	771.01	**742.55**	771.96	566.25	**489.68**	565.63	595.77	472.04	**465.78**	732.94	**422.21**	426.78	953.14	**394.44**	441.96	1087.17	**385.23**	492.94
	Prefill	7.62	**0.23**	7.63	5.71	**0.34**	5.67	6.05	**0.44**	4.39	7.58	0.62	**3.84**	10.19	**0.86**	3.9	11.79	**1.63**	4.47
	TTFT	778.63	**742.78**	779.58	571.96	**490.02**	571.31	601.82	472.47	**470.17**	740.53	**422.83**	430.62	963.33	**395.3**	445.86	1098.96	**386.85**	497.41
codeU	Queuing	409.95	423.25	**371.25**	464.0	378.41	**308.82**	610.87	**299.07**	315.34	749.75	**278.63**	352.16	812.07	**258.87**	424.72	993.57	276.36	**232.95**
	Prefill	9.32	**0.4**	7.53	10.91	**0.53**	5.15	14.82	**0.59**	4.6	19.35	**0.7**	4.82	22.86	**0.93**	5.7	21.2	**1.77**	3.79
	TTFT	419.27	423.65	**378.78**	474.92	378.94	**313.97**	625.7	**299.56**	319.94	769.11	279.33	**356.98**	834.94	**259.8**	430.42	1014.77	278.14	**236.74**
shareGPT	Queuing	-	-	-	**112.19**	165.75	113.31	106.09	149.44	**103.49**	104.4	156.95	**102.44**	104.32	147.19	**101.28**	103.8	162.25	**101.36**
	Prefill	-	-	-	0.16	**0.12**	0.15	0.17	0.17	**0.16**	0.2	0.28	0.2	0.25	0.28	**0.24**	0.32	0.48	**0.31**
	TTFT	-	-	-	**112.34**	165.87	113.46	106.26	149.61	**103.65**	104.6	157.23	**102.64**	104.57	147.19	**101.52**	104.12	162.72	**101.67**

speedup, and up to 3.64× and 8.34× TPOT speedup over vLLM on NarrativeQA and codeU, respectively. Compared to Ranking, *PKAS* delivers over 5× TPOT speedup on NarrativeQA and codeU, and over 1.6× on ShareGPT across all chunk sizes.

Table 2 reports queuing time, prefill time, and TTFT across these datasets. Compared to Ranking, *PKAS* achieves lower TTFT primarily through reduced queuing time, while reducing both queuing and prefill time relative to vLLM.

5.3 Performance Results on Multiple GPUs

In this experiment, we evaluate *PKAS* against other baselines at scale using the Yi-Coder-9B-Chat-8x-MoE model to analyze the performance implications of cross-GPU communication due to tensor-parallelism (4 GPUs). We configured vLLM with a maximum batch size of 1024 requests to ensure triggering the KV cache overflow when using the ShareGPT dataset. Other parameters are identical to the single-GPU setup.

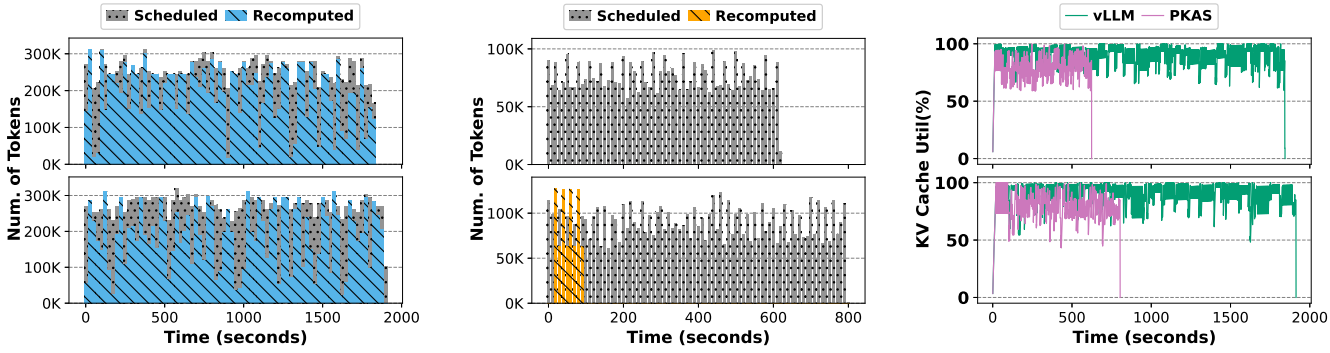
Throughput and Total Recomputed Tokens. Figure 8 shows the throughput and total recomputed tokens across these three datasets. As can be seen, our approach yields consistently higher throughput and significantly lower recomputation overhead caused by KV cache overflow compared to vLLM and Ranking. For NarrativeQA (Figure 8a), our approach outperforms vLLM by up to 2.64× at intermediate chunk sizes (2K–16K) and Ranking by up to 1.49× at 2K chunk size. codeU (Figure 8b) exhibits even greater throughput improvement, where *PKAS* achieves up to 3.91× and 1.81× over vLLM and Ranking respectively. On ShareGPT (Figure 8c), where recomputation overhead is inherently lower, *PKAS* still achieves approximately 3% higher throughput than vLLM and about 1.61× speedup over Ranking. These findings highlight *PKAS*'s effectiveness on MoE models, achieving substantial gains for long-context requests and moderate improvements for short-context ones. Similar to the single GPU case, these improvements are driven by *PKAS*'s ability to minimize recomputations during inference serving (shown on the minor y-axis).

Latency. The prefill time and TPOT latency for the MoE model are reported in Figure 9. Table 2 further provides a breakdown of TTFT into queuing and prefill time when running with this model across different datasets and chunk sizes, revealing which stage contributes most to TTFT latency. Overall, our approach achieves lower and more stable TTFT and TPOT latency compared to vLLM across all configurations. For the NarrativeQA and codeU datasets, we significantly reduced both TTFT and TPOT latency relative to vLLM, particularly at chunk sizes between 2K and 16K tokens. Compared with Ranking, although *PKAS* exhibits a higher prefill time on long context datasets, it compensates with significantly lower queuing time and TPOT latency, yielding better overall performance. These results showcase that *PKAS* can effectively reduce the end-to-end latency by minimizing queuing delays and maintaining low decoding overhead.

5.4 Ablation: Characterizing Recomputation Overhead and KV Cache Utilization

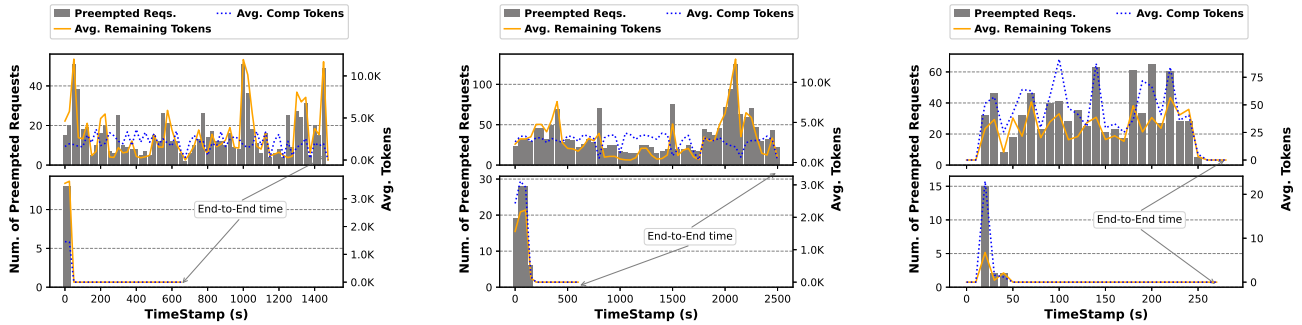
To further characterize the performance, we perform an ablation study using NarrativeQA with an 8K chunk size to examine two aspects: 1) the number of scheduled tokens versus recomputed tokens over time; and 2) the KV cache utilization over time. These tests evaluate both single-GPU Llama-3.1-8B and the 4-GPU-based Yi-Coder-9B-Chat-8x-MoE models.

Figure 10a and Figure 10b present the number of scheduled and recomputed tokens when running with vLLM vs *PKAS* for the 1 GPU 8B model (top) and 4 GPU MoE model (bottom). vLLM (Figure 10a) schedules a large number of tokens compared to *PKAS* (observed by varying limits of y-axis) and also triggers substantial recomputations (shown by non-gray bars) for both the models. This is because vLLM uses a greedy strategy, as described in § 3, which leads to more recomputation overhead in the subsequent iterations (blue bars in Figure 10a). Consequently, *PKAS*'s prediction-based look-ahead admissibility significantly mitigate recomputations (orange bars in Figure 10b). While *PKAS* nearly eliminates all recomputations for the 1 GPU Llama-3.1-8B model (top), we observe that it shows some recomputations with the 4 GPU Yi-Coder-9B-Chat-8x-MoE (bottom) because in the initial iterations, when no requests have completed, *PKAS* uses a default output length of 1 and behaves similarly to vLLM's aggressive scheduling. However, despite the few recomputations, *PKAS* still leads to faster computations– vLLM



(a) vLLM (b) PKAS (c) KVCache Utilization

Figure 10: Comparison of: 1) Number of scheduled and recomputed tokens over time ((a) vLLM and (b) *PKAS*); and 2) KV Cache utilization over time (higher utilization while shorter completion time is better).(dataset: NarrativeQA chunk size: 8K, Llama-3.1-8B(top) and Yi-Coder-9B-Chat-8x-MoE(bottom))



(a) NarrativeQA (b) CodeU (c) ShareGPT

Figure 11: Preemption analysis: vLLM(top) vs. PKAS(bottom) over time across different datasets, showing the number of preempted requests (gray bars), average computed tokens (blue dotted), and average remaining tokens (orange) per preemption event. (chunk size: 4K; Llama-3.1-8B model; fewer preempted requests and shorter end-to-end time is better)

takes ~2000s vs *PKAS* takes only 600s on the 1 GPU Llama-3.1-8B model and 800s on the 4 GPU Yi-Coder-9B-Chat-8x-MoE.

Figure 10c shows how KV cache utilization evolves over time when using *PKAS* vs. vLLM. As can be seen, *PKAS* exhibits slightly lower KV cache utilization (ranging from 50% to 95% compared to vLLM (ranging from 70% to 100%), but it achieves a substantial reduction in end-to-end execution time for both models (e.g., for Llama-3.1-8B, it decreases from 1843s to 623s; for the MoE model, it decreases from 1913s to 803s). The slightly low utilization is due to *PKAS*'s admissibility control maintaining headroom to prevent overflow, which avoids costly preemptions that negate the benefit of higher utilization. This experiment confirms that our approach can achieve dual benefits: improved throughput and sustained high KV cache utilization. Thanks to *PKAS*'s optimistic output length prediction with an efficient deviation-aware moving average method, it effectively balances the trade-off between recomputation and KV cache utilization.

5.5 Ablation: Analyzing Preemption Behavior

Next, we compare and analyze the preemption behavior over time between vLLM and *PKAS* using the Llama-3.1-8B model across three different datasets. As shown in Figure 11, vLLM (top) exhibits frequent preemption events throughout inference serving across all workloads. Furthermore, to zoom on the circumstances under which the preemptions happen, we also depict the average number of tokens computed so far and the average number of tokens remaining for the preempted requests. The high number of preempted requests correlates with the high number of recomputed tokens discussed in § 5.4 and supports the observation that vLLM's aggressive scheduling may lead to preempting requests that have already made significant progress (high number of average computed tokens), resulting in wasted computation. In contrast, *PKAS* (bottom) presents far fewer preemption events, which occur primarily at the beginning of execution and remain near zero thereafter. These early preemptions result from our moving average predictor's cold-start phase, where insufficient completion history causes *PKAS* to temporarily adopt vLLM-like aggressive scheduling. However, even during cold-start, the average computed and remaining tokens per preemption are substantially lower than vLLM, indicating minimal progress loss and recomputation overhead caused by preempted requests. As requests complete and the predictor stabilizes, *PKAS* effectively eliminates further preemptions. Also, in the case of vLLM (top), spikes in the number of preemptions tend to correlate with spikes in computed/remaining tokens because vLLM preempts the

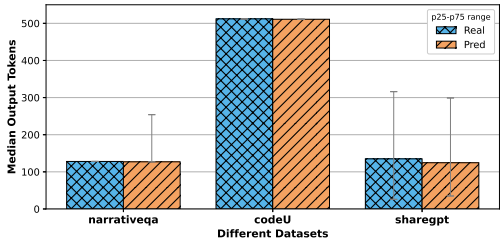


Figure 12: Median predicted vs. real output length across datasets; error bars show p25–p75 range (Llama-3.1-8B, chunk size: 8k)

most recently admitted requests, which have the most remaining tokens. This explains why the number of recomputed tokens remains high (as observed in Figure 10a).

5.6 Ablation: Prediction Accuracy and Impact

Figure 12 compares the median predicted output length against the actual median across all three datasets spanning long-context and short-context workloads, with error bars showing the interquartile range (IQR, p25–p75 representing the middle 50% of outputs). We report the interquartile range rather than min–max because the output-length distributions are long-tailed. Min–max is dominated by outliers, while p25–p75 is robust to such tails. Furthermore, since the output length predictor biases by one standard deviation below the mean, IQR helps visualize the uncertainty the predictor is designed to adapt to. For NarrativeQA and codeU, the output lengths are tightly distributed (small standard deviation), so the one standard deviation bias is small and the predictor matches the real median almost exactly with low variance. The asymmetric upper error bar on NarrativeQA's predicted output length reflects a right-skewed prediction; most predictions still sit at the median (p25 ≈ median, hiding the lower whisker behind the bar), while a small fraction extends upward to lift p75 above the median. On ShareGPT, real outputs show a much wider interquartile range, yet the predicted output lengths still stay close to the real median despite the wider spread. Our predictor remains reliable under such variability because its statistics update online with every completed request and the Simulator re-evaluates admissibility at each forward pass, so transient mispredictions are corrected before they accumulate into preemptions. Thus, even with high-variance ShareGPT, *PKAS* achieves near-zero preemptions after a brief cold-start (Figure 11c) and a modest throughput gain over vLLM, the binary admission gate and the uncertainty-adaptive bias enable efficient admission across workloads without degrading performance.

6 Related Work

LLM Serving Engines. Contrary to conventional DNN serving systems (e.g.,Nvidia Triton [24] and FastTransformer [23]) that use fixed-sized batches, runtimes like Orca [39] propose continuous batching, wherein requests can join or leave a batch at per iteration. To further reduce stalls caused by large prefills, Sarthi-Serve [1] and DeepSpeed-FastGen [14] proposed chunked prefilling and coalesced batching of prefill and decode requests, which is widely adopted in state-of-the-art inferencing engines such as vLLM [20], SGLang [40], TensorRT-LLM [25], etc. These optimizations improve serving throughput but introduce challenges pertaining to fine-grained iteration-level co-scheduling of prefill and decode batches under limited memory budgets, which *PKAS* addresses.

Scheduling Policies. There has been extensive research about optimizing scheduling policies for different use cases across HPC, such as FCFS, earliest-deadline-first (EDF), shortest-job-first (SJF), and deep learning specific approaches [38]. However, inference requests batched in each forward pass might produce arbitrary output length [32], have stringent deadlines based on service-level objectives (SLOs) [16], and consume variable KV cache [37], those conventional scheduling strategies cannot be applied directly. Recent works, such as S3 [19], learning-to-rank(LRT) [10], and SSJF [27], have attempted to predict generation length using small predictor models (i.e., DistilBERT and OPT) to guide scheduling decisions. LTR will rank the requests based on their output length to mitigate the head-of-line blocking (HOL) caused by FCFS (a widely used and default scheduling approach of state-of-the-art serving engines). However, these methods require training or fine-tuning a dedicated predictor, adding extra overhead and limiting adaptability to diverse workloads. Instead, *PKAS* uses a simple mathematical formula for prediction, removing the dependency on auxiliary models. Furthermore, disaggregated prefill/decode architecture has gained traction in multi-GPU environments, leading to new scheduling approaches tailored for such setups, such as Arrow [35], Llumnix [29], TetriInfer [15]. Our approach is orthogonal to these methods, and our fast-forward KV cache simulation mechanism can be integrated with such techniques. We are the first to reveal significant bottlenecks due to a ping-pong pattern of unnecessary preemptions and resumptions, and to solve them with a lightweight KV cache simulation technique.

7 Conclusions

This work proposes *PKAS*, a predictive KV cache-aware batch scheduling approach to accelerate transformer inferences by reducing preemptions. By isolating predictions as admissibility recommendations of new inference request candidates, *PKAS* can be integrated with diverse scheduling approaches such as continuous batching and chunked prefill. We showcase the integration of our approach in vLLM. Our evaluation across models under diverse real-world workloads presents that *PKAS* can achieve up to ≈7.34x higher throughput and ≈8x lower TOPT over vLLM, while outperforming Ranking by ≈2.19x in throughput and ≈5x in TPOT. In future work, we will consider the integration of our approach with several other batching strategies, including disaggregated strategies that run the prefill and decode phases on different inference instances. Furthermore, we will extend *PKAS* to support more complex output-length prediction techniques, hiding their by proactively running them during the forward passes.

Acknowledgments

This work is supported in part by the U.S. Department of Energy (DOE), Office of Advanced Scientific Computing Research (ASCR) under contract DEAC02–06CH11357/0F–60169, and the National Science Foundation (NSF), Office of Advanced Cyberinfrastructure, under grants Core-2313154, CSSI-2411318, CSSI-2104013, CSSI-2411386/241387, CSSI-2514056 and 2106635.

References

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

[2] 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)*. Association for Computational Linguistics, Bangkok, Thailand, 14388–14411.

[3] anon8231489123. 2023. *ShareGPT_Vicuna_unfiltered*. Hugging Face. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/tree/main

[4] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2024. Longbench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*. Association for Computational Linguistics, Bangkok, Thailand, 3119–3137.

[5] Gantavya Bhatt, Yifang Chen, Arnav Das, Jifan Zhang, Sang Truong, Stephen Mussmann, Yinglun Zhu, Jeff Bilmes, Simon Du, Kevin Jamieson, et al. 2024. An experimental design framework for label-efficient supervised finetuning of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, Bangkok, Thailand, 6549–6560.

[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. arXiv:2303.12712 [cs.CL] https://arxiv.org/abs/2303.12712

[7] Daniel Castro. 2024. Rethinking Concerns About AI's Energy Use. https://www2.datainnovation.org/2024-ai-energy-use.pdf

[8] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. arXiv:2507.06261 [cs.AI] https://arxiv.org/abs/2507.06261

[9] Hugging Face. 2023. *Text Generation Inference*. Hugging Face. https://github.com/huggingface/text-generation-inference

[10] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient llm scheduling by learning to rank. *Advances in Neural Information Processing Systems* 37 (2024), 59006–59029.

[11] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. 2025. Niyama: Breaking the Silos of LLM Inference Serving. arXiv:2503.22562 [cs.DC] https://arxiv.org/abs/2503.22562

[12] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

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

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

[16] Azam Ikram, Xiang Li, Sameh Elnikety, and Saurabh Bagchi. 2025. Ascendra: Dynamic Request Prioritization for Efficient LLM Serving. arXiv:2504.20828 [cs.AI] https://arxiv.org/abs/2504.20828

[17] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[18] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. 2025. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *ACM Transactions on Computer Systems* 44, 1 (2025), 1–27.

[19] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: Increasing GPU Utilization during Generative Inference for Higher Throughput. *Advances in Neural Information Processing Systems* 36 (2023), 18015–18027.

[20] 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*. Association for Computing Machinery, Koblenz, Germany, 611–626.

[21] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[22] Benjamin S Manning, Kehang Zhu, and John J Horton. 2024. *Automated social science: Language models as scientist and subjects*. Technical Report. National Bureau of Economic Research.

[23] Nvidia. 2024. *FastTransformer*. NVIDIA Corporation. https://github.com/NVIDIA/FasterTransformer

[24] NVIDIA. 2024. *NVIDIA Triton Dynamic Batching*. NVIDIA Corporation. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_configuration.html#dynamic-batcher

[25] NVIDIA. 2024. *TensorRT-LLM*. NVIDIA Corporation. https://github.com/NVIDIA/TensorRT-LLM Accessed: 2026-04-30.

[26] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of machine learning and systems* 5 (2023), 606–624.

[27] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Efficient Interactive LLM Serving with Proxy Model-Based Sequence Length Prediction. arXiv:2404.08509 [cs.DC] https://arxiv.org/abs/2404.08509

[28] Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. 2025. Don't Stop Me Now: Embedding-Based Scheduling for LLMs. In *The Thirteenth International Conference on Learning Representations (ICLR '25)*. OpenReview.net, Singapore. https://openreview.net/forum?id=7JhGdZvW4T

[29] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, USA, 173–191.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017), 5998–6008.

[31] vllm. 2024. *A high-throughput and memory-efficient inference and serving engine for LLMs*. vLLM Project. https://github.com/vllm-project/vllm/tree/main/vllm

[32] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. arXiv:2401.17644 [cs.DC] https://arxiv.org/abs/2401.17644

[33] B. P. Welford. 1962. Note on a method for calculating corrected sums of squares and products. *Technometrics* 4, 3 (1962), 419–420.

[34] 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*. Association for Computing Machinery, Austin, TX, USA, 640–654.

[35] Yu Wu, Tongxuan Liu, Yuting Zeng, Siyu Wu, Jun Xiong, Xianzhe Dong, Hailong Yang, Ke Zhang, and Jing Li. 2025. Arrow: Adaptive Scheduling Mechanisms for Disaggregated LLM Inference Architecture. arXiv:2505.11916 [cs.DC] https://arxiv.org/abs/2505.11916

[36] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[37] Jie Ye, Jaime Cernuda, Avinash Maurya, Xian-He Sun, Anthony Kougkas, and Bogdan Nicolae. 2025. Characterizing the behavior and impact of kv caching on transformer inferences under concurrency. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, Milan, Italy, 1191–1202.

[38] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2024. Deep Learning Workload Scheduling in GPU Datacenters: A Survey. *ACM Comput. Surv.* 56, 6, Article 146 (Jan. 2024), 38 pages.

[39] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, USA, 521–538.

[40] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[41] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2023. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems* 36 (2023), 65517–65530.

[42] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, et al. 2025. {NanoFlow}: Towards Optimal Large Language Model Serving Throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, USA, 749–765.