Priority-Aware Preemptive Scheduling for Mixed-Priority Workloads in MoE Inference

Mohammad Siavashi
KTH Royal Institute of Technology
Stockholm, Sweden

Faezeh Keshmiri Dindarloo[*]
Unaffiliated Researcher
Stockholm, Sweden

Dejan Kostić
KTH Royal Institute of Technology
Stockholm, Sweden

Marco Chiesa
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

Large Language Models have revolutionized natural language processing, yet serving them efficiently in data centers remains challenging due to mixed workloads comprising latency-sensitive (LS) and best-effort (BE) jobs. Existing inference systems employ iteration-level first-come-first-served scheduling, causing head-of-line blocking when BE jobs delay LS jobs. We introduce QLLM, a novel inference system designed for Mixture of Experts (MoE) models, featuring a fine-grained, priority-aware preemptive scheduler. QLLM enables expert-level preemption, deferring BE job execution while minimizing LS time-to-first-token (TTFT). Our approach removes iteration-level scheduling constraints, enabling the scheduler to preempt jobs at any layer based on priority. Evaluations on an Nvidia A100 GPU show that QLLM significantly improves performance. It reduces LS TTFT by an average of 65.5× and meets the SLO at up to 7 requests/sec, whereas the baseline fails to do so under the tested workload. Additionally, it cuts LS turnaround time by up to 12.8× without impacting throughput. QLLM is modular, extensible, and seamlessly integrates with Hugging Face MoE models.

CCS Concepts

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

Keywords

Large Language Models, Mixture-of-Experts, Preemptive Scheduling, Latency-Sensitive Inference, GPU Acceleration, Priority-Aware Scheduling

ACM Reference Format:
Mohammad Siavashi, Faezeh Keshmiri Dindarloo, Dejan Kostić, and Marco Chiesa. 2025. Priority-Aware Preemptive Scheduling for Mixed-Priority Workloads in MoE Inference. In *Proceedings of 5th Workshop on Machine Learning and Systems (EuroMLSys '25)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3721146.3721956

1 Introduction

Large Language Models (LLMs) have significantly advanced the domain of Natural Language Processing (NLP), enabling tasks such as machine translation, summarization [16, 20, 23], code synthesis and completion [4], and conversational AI [1, 22]. The Mixture of Experts (MoE) architecture [11], a transformer variant that selectively activates subsets of specialized feedforward layers per token, has emerged as the preferred paradigm for large-scale models that can attain superior performance while ensuring rapid inference. Although the model in its entirety can be extensive (e.g., 671 B parameters for DeepSeek-R1 [6]), the selective activation of so-called experts can substantially reduce inferencing costs and subsequently enhance adoption [13].

Data centers serving LLMs typically handle two types of requests: high priority, which are *latency-sensitive (LS)*, and low priority, which are *best-effort (BE)* [26, 28]. High priority requests might originate from users with paid subscriptions that include a *Service Level Objective* (SLO) agreement or from interactive applications like ChatBots. In contrast, BE requests could come from users on free tiers or throughput-oriented jobs such as document summarization [28]. Consequently, inference systems must identify and differentiate between LS and BE requests, ensuring low *time-to-first-token (TTFT)* and quick turnaround time for LS jobs while maintaining high throughput for BE jobs.

Current large-scale inference systems for LLMs, such as Orca [10], vLLM [27], and Hugging Face (HF) TGI [7], predominantly employ *iteration-level scheduling*, where new jobs are incorporated, and completed jobs are removed only at the end of each iteration. While this approach enables efficient batching, it adheres to a *first-come-first-served (FCFS)* strategy, treating all inference jobs equally and failing to prioritize LS jobs over BE workloads. As a result, LS jobs frequently experience *head-of-line (HOL) blocking* [12], where large BE jobs with long input and output sequences monopolize resources, delaying LS execution. These delays are particularly pronounced in LLM inference workloads, where request sizes vary significantly, exacerbating scheduling inefficiencies. Addressing this issue requires an inference system capable of distinguishing between LS and BE jobs, reacting to LS job arrivals with minimal delay, and enabling *fine-grained preemption* of BE computations to improve overall system responsiveness.

The dynamic top-k token routing inherent to MoE architectures necessitates granular state management: preempted sequences must retain not only their KV cache, but also expert assignments, routing metadata, and partial computations of the k selected experts

[*] Work done while at KTH Royal Institute of Technology

EuroMLSys '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1538-9/25/03
https://doi.org/10.1145/3721146.3721956

to ensure deterministic resumption [25]. Fine-grained preemptive scheduling is becoming increasingly feasible with modern hardware advancements, such as NVLink's low-latency interconnects [19] and unified memory architectures [18], which are becoming increasingly prevalent in data centers. However, these advancements also require specialized scheduling mechanisms tailored to contemporary MoE models.

This paper introduces QLLM, an inference system that reduces LS job latency in MoE models through fine-grained preemption and priority-aware scheduling at the expert level. QLLM features: (1) a redesigned MoE layer with per-expert queues for dynamic buffering and low-overhead state management, and (2) a priority-aware scheduler that mitigates HOL blocking by distinguishing LS and BE jobs. Unlike existing inference systems using iteration-level execution, QLLM allows independent expert processing, allowing LS jobs to preempt BE jobs without discarding intermediate computations. An efficient state management mechanism preserves execution progress, allowing seamless BE resumption. The scheduler optimizes LS latency while maintaining high throughput.

Our evaluation on an Nvidia A100 80 GB GPU shows that QLLM reduces TTFT by up to 101.6× (avg. 65.2×), enabling SLO compliance for up to 7 jobs per second. QLLM maintains comparable or higher throughput than existing systems and reduces LS turnaround time by up to 12.8×.

Our work makes the following contributions:

Novel MoE Layer Design: We introduce per-expert queues to enable token buffering and deferred execution, eliminating rigid layer-wise synchronization. This design allows independent expert execution, enhancing scheduling flexibility.

Priority-Aware Scheduler: QLLM incorporates a scheduler that differentiates LS and BE jobs, ensuring low-latency scheduling and efficient GPU resource allocation.

Fine-Grained Expert-Level Preemption: QLLM enables BE job preemption at the expert level, reducing LS job queuing delays. This is achieved via a lightweight state management mechanism, a unified KV cache abstraction for batch updates, and per-expert queuing.

Real-World Evaluation: We evaluate QLLM on real hardware with Mixtral 8×7B, demonstrating improved LS job latency while maintaining high throughput.

Modular and Extensible Framework: QLLM integrates seamlessly with Hugging Face MoE models with minimal modifications (e.g., class inheritance), facilitating deployment, extensibility, and further research in MoE inference.

This paper discusses our initial findings, a preliminary evaluation, and limitations. Our ultimate plan is to release QLLM as an open-source project in future versions.

2 Background and Motivation

2.1 Mixture of Experts Models

MoE models are a type of transformer model [29] designed to enhance computational efficiency in LLMs by selectively activating only a subset of parameters during inference. Unlike dense transformer models, which process all tokens through fully activated feedforward layers, MoE replaces these layers with multiple expert networks and a router that assigns each token to the most relevant experts. This selective activation reduces computational overhead while preserving the overall capacity of the model [13, 25]. State-of-the-art MoE models, such as Mixtral [11], OpenAI GPT-4 [3], and DeepSeek v3 [5], exemplify this architecture.

2.2 Prefill and Decode Phases

In an LLM, each transformer layer's self-attention mechanism determines token interactions using key (K) and value (V) tensors. To generate new tokens efficiently, the model stores KV pairs of all previous tokens in a KV cache, reducing redundant computation and improving inference speed [14, 15]. The output of self-attention is then passed to a per-layer router, which selects experts responsible for generating the final output.

LLM inference consists of two phases: *prefill* and *decode*. Prefill processes input tokens in parallel, generating KV cache entries while producing a single output token. Decode operates iteratively, generating one token at a time while leveraging and updating the KV cache. Prefill is compute-intensive due to self-attention across all tokens, whereas decode is memory-intensive as it computes self-attention only between the new token and previous ones [2, 10, 33].

2.3 Challenges in Existing Inference Systems

Modern LLM inference systems employ iteration-level scheduling [10, 17, 27], where a batch of jobs (*i.e.*, prompts) is processed together, generating one token per job per iteration. An iteration corresponds to a full execution of all model layers. Building upon this, continuous batching, as implemented in vLLM and HF TGI, dynamically updates batch composition at every iteration. It removes completed jobs and adds new ones to maintain batch efficiency. If new jobs arrive and a decode batch has available space, the scheduler stops decoding, prefills the new arrivals, and extends the batch for the next decode iteration [8]. Then, the scheduler runs the decode batch in a run-to-completion fashion, meaning iterations continue until the full response of a job is generated.

Existing inference engines typically pad and concatenate input tensors from multiple jobs into a single tensor before execution. Although this improves computation efficiency in run-to-completion systems, it complicates the isolation and updating of individual jobs within inner layers. Since model layers only see low-level tensors, oblivious to corresponding sequences, tracking state updates for each sequence becomes inherently difficult and expensive; thus, state updates for sequences occur only at the iteration level.

For instance, if such a layer-level scheduler operates with a defined policy that preempts tasks based on memory limitations in inner layers, it necessitates the extraction of the token from the running batch tensors. This extraction demands costly structural transformation operations on the tensors to accurately retrieve and preserve the token's state such as kv cache entries, attention mask, hidden states, residuals, routing information, and associated metadata. Additionally, when restoring a preempted token, the system must not only reload its exact state, but also reconfigure its tensors (*e.g.*, by padding the KV cache) to ensure seamless integration with the current running batch, a process susceptible to shape mismatches. Moreover, since current inference systems do not track individual sequences within inner layers (*i.e.*, models only see tensors internally), dynamically modifying batch composition can disrupt data flow, resulting in outputs that no longer align with their corresponding inputs. Such challenges render traditional inference systems impractical for achieving fine-grained scheduling at the layer or expert level.

2.4 Limitations of Preemption in Current Systems

Current inference systems are priority-oblivious. This, combined with the run-to-completion approach of the schedulers, causes delays for LS jobs as they await the completion of time-consuming BE jobs. As a result, the queueing time for LS jobs increases, considerably prolonging TTFT and *turnaround time*, the duration from when a job enters the system until its response is fully generated. This issue is known as HOL blocking in the scheduling context, where extended processing of BE jobs influences the latency of LS jobs [12, 32].

In iteration-level scheduling, there exists a potential for systems to decide on preemption at the granularity of token generation [31]. Nevertheless, the increasing number of layers and the strong inter-layer dependencies inherently introduce significant delays until the subsequent iteration. In our experiments, conducted on the A100 with the Mixtral 8x7B model, each decode iteration requires 300-400 ms. For instance, if a BE job is processing at the first layer when an LS job arrives, the entire iteration execution time will be appended to the LS job's TTFT until it has the opportunity to be scheduled in the next iteration.

An efficient scheduling strategy should have low overhead and support fine-grained job preemption and context switching. It also needs to dynamically prioritize LS jobs to reduce TTFT without significantly affecting system throughput. A strong solution should meet these challenges with minimal model changes, ensuring compatibility across frameworks.

3 QLLM Design Overview

Effectively handling mixed-priority workloads requires rethinking the scheduling and inference engine components with the goal of enabling rapid preemption of jobs and context switching at the granularity of experts.

3.1 Challenges and Design Decisions

We set the following objectives when developing QLLM.

Generic and low overhead preemption mechanism. Fine-grained control of the system state (*e.g.*, KV caches, hidden states) within the inner layers and orchestration of the execution flow is a generic need in LLM serving systems. We use a centralized state management mechanism for sequences and batches that enables real-time tracking and updating of sequence states within layers, avoiding delayed updates at the iteration level.

Minimizing LS Job latency without sacrificing throughput. The key challenge is reacting to high-priority workloads with fine-grained timing. To address this, QLLM implements a priority-aware scheduler working in tandem with a low-overhead preemption mechanism, enabling scheduling decisions at each layer to optimize LS task responsiveness while maintaining efficient execution flow.

Unified sequence and batch management. In contrast to current systems that keep concatenated batch tensors throughout response generation, a system such as QLLM requires manipulating the batch within inner layers due to context switching. Therefore, we designed a *unified lifecycle management abstraction*, which encapsulates all sequence-associated states and metadata into a sequence and batch framework. This framework simulates a single batch tensor to ensure compatibility with existing models, while also allowing asynchronous updates to each sequence's individual tensors using a facade pattern abstraction.

User-defined scheduling policies. Tailored scheduling policies specific to the workload can optimize performance in various systems. QLLM facilitates user-defined scheduling policies (in Python) at the expert level by employing a checkpointing mechanism alongside a closed-loop controller system.

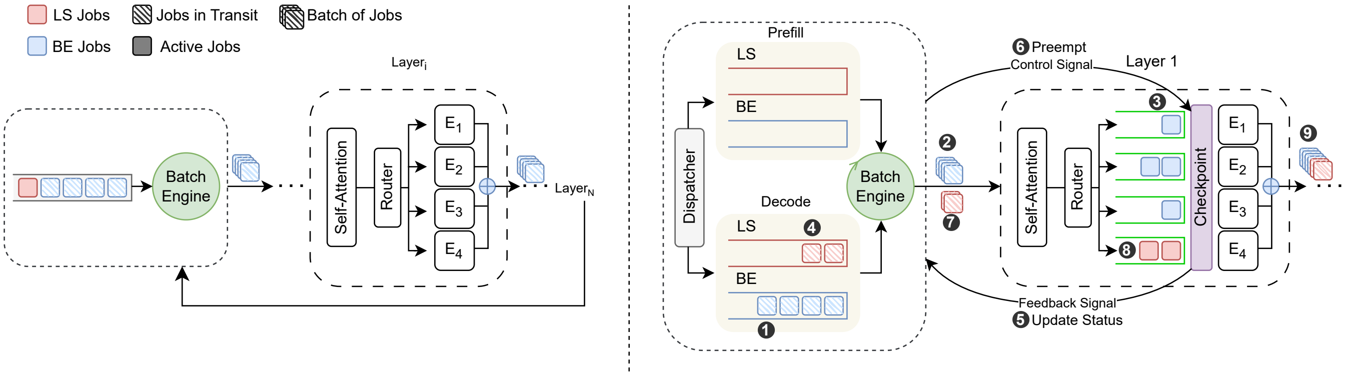
Top-k expert selection. Top-k expert selection for each token is an additional challenge for preemptive schedulers. To maximize QLLM's flexibility for user-defined policies, we need a solution that enables partial processing with preemption while avoiding queue stalls due to delayed experts. Our per-expert queuing and single source of truth for state management make this efficient by pushing sequence references into multiple queues and tracking outputs as state. QLLM distinguishes fully from partially processed tokens, ensuring only complete tokens contribute to the hidden state of the next layer.

Modular architecture. QLLM addresses architectural bottlenecks in systems like vLLM [30] by adopting a modular and extensible design. Through *modularization*, *encapsulation*, and *dependency injection*, it decouples the system logic, minimizing interdependencies, and enabling seamless policy updates.

3.2 System Architecture

Figure 1 illustrates the architecture of QLLM. At a high level, a *scheduler* component receives a sequence of prompts over time—each referred to as a *job*—and schedules them on the *inference engine*, which processes jobs through the layers of the model. The scheduler is architected around two primary components: a *dispatcher* and a *batch engine*. The dispatcher enqueues incoming jobs into prefill queues based on priority and directs model output tokens into their corresponding decode queues. Simultaneously, the batch engine groups tokens into batches for each iteration, following Algorithm 1, before dispatching them to the inference engine. Unlike existing work, the QLLM scheduler enables *preemption & scheduling of jobs at fine granularity*. For example, QLLM allows preempting a batch of best-effort jobs at any processing layer to replace one of the jobs with a newly arrived latency-sensitive job, then resuming their execution.

The QLLM scheduler utilizes four separate queues to monitor unfinished LS and BS jobs. Two of these queues manage jobs that need to be processed through the prefill stage, while the other two handle jobs that have progressed to the decode phase. Queues are implemented with an FCFS policy. The batch engine in the scheduler extracts jobs from these queues to implement a pre-configured (or a user-defined) batch policy. We show the pre-configured policy in Algorithm 1 where we prioritize the execution of LS jobs by



(a) Baseline

(b) QLLM

Figure 1: Comparison of a baseline and QLLM. The baseline employs iteration-level scheduling and continuous batching, with control returning to the scheduler only upon execution of all N layers. The figure on the right demonstrates a streamlined execution of QLLM's fine-grain scheduling within layer 1. LS jobs arrive after BE jobs and are batched in step 7.

Algorithm 1 Batch Selection Logic

```
1:  function GetNextBatch
2:      if LS_DecodeQueue.size() ≥ BatchSize then
3:          return GetBatch(LS_DecodeQueue)
4:      else if not LS_PrefillQueue.isEmpty() then
5:          batch ← GetBatch(LS_PrefillQueue)
6:          if batch.size() < BatchSize then
7:              Fill from BE_PrefillQueue
8:          return batch
9:      else if not LS_DecodeQueue.isEmpty() then
10:         batch ← GetBatch(LS_DecodeQueue)
11:         if batch.size() < BatchSize then
12:             Fill from BE_DecodeQueue
13:         return batch
14:     else if not BE_DecodeQueue.isEmpty() then
15:         return GetBatch(BE_DecodeQueue)
16:     else if not BE_PrefillQueue.isEmpty() then
17:         return GetBatch(BE_PrefillQueue)
18:     return None
```

preempting BE jobs. Batches are then submitted to the inference engine.

QLLM allows for user-defined scheduling and context-switch policies at the granularity of experts, which can be implemented in fewer than 50 lines of Python code. The inference engine operates as a closed-loop feedback controller, updating the scheduler on execution status after each attention and router stage. In response, the scheduler dynamically signals the engine to adapt the execution flow based on user-defined policies. While in this paper we define our policies to minimize TTFT for LS tasks, other systems could leverage QLLM to define policies for dynamic sequence offloading or selective expert execution based on user-defined constraints, showcasing its extensibility.

Example. In Figure 1, four BE jobs must be processed at the decode stage. Both the baseline (Fig. 1(a)) and QLLM (Fig. 1(b)) batch and dispatch these jobs to the inference engine. Slightly after, an LS job arrives. With continuous batching, the baseline approach waits until all the BE jobs are processed through all the layers. Only then, it executes the prefill phase of the LS job, and adds it to the batch. In QLLM, as soon as an LS job arrives, the scheduler stops the execution of the BE batch at the engine. It then executes the prefill stage for the LS job, and then the subsequent decode stage. Then, QLLM dynamically merges the LS jobs with the BE jobs within the layer execution, reducing the latency of the LS job.

Batch selection logic. Now we explain how Algorithm 1 creates batches. The goal is to fill a batch with LS jobs without exceeding the maximum batch size. The algorithm prioritizes filling the batch first with LS jobs at the decode stage (lines 2-3). If a maximum-sized batch cannot be created, it prioritizes LS jobs in the prefill stage to produce more LS jobs in the decode stage (lines 4-5). The algorithm also attempts to add some BE jobs to fill the batch if possible (lines 6-7). If there are no LS jobs at the prefill stage, the algorithm simply executes the LS jobs at the decode stage, filling the batch with BE jobs at the decode stage (lines 10-14). If there are no LS jobs, it simply executes BE jobs, prioritizing decode over prefill (lines 16-19).

3.3 System Modularity and Extensiblity

Unified sequence and batch abstractions for inference. At its core, QLLM replaces the fragmented sequence and batch handling found in existing inference systems with a unified execution model. Rather than managing sequences and their state in an ad hoc manner, QLLM defines a Sequence abstraction, encapsulating all relevant metadata, including KV cache tensors, routing information, and execution state. A corresponding Batch abstraction allows collective processing while maintaining individual sequence integrity. This structured approach streamlines execution, improves observability, and ensures robust preemption without introducing unnecessary synchronization overhead.

Breaking rigid batch processing with queues per expert. A major advancement is the per-expert FIFO queuing, which fundamentally transforms the token flow through MoE layers. In contrast to traditional inference engines that create rigid batch tensors retained across iterations until execution completion, QLLM employs distinct and independent data structures-including tensors for each Sequence object-and offers a unified interface through abstraction, ensuring compatibility without necessitating destructive modifications.

This is achieved through the *Facade Pattern*, where the *Batch* abstraction acts as an interface between the model and underlying sequence-level data structures. Instead of handling anonymous concatenated tensors, QLLM provides a structured representation where each *Sequence* object retains its own state while being processed within a unified *Batch*. This allows the model to interact with monolithic tensors while the underlying system dynamically tracks and updates individual sequences upon tensor modification. By intercepting tensor access, QLLM enables fine-grained control over execution flow, facilitating expert-level preemption and efficient context switching without modifying core model operations.

Eliminating costly split-merge procedures. Unlike existing commercial systems where retrieving a token from a batch involves costly splitting and concatenating of pre-constructed tensors, QLLM enables direct, real-time state updates of individual sequences, facilitating low-overhead preemption by overlapping token state updates with execution. Additionally, QLLM introduces efficient KV cache management through a novel module which we call the Unified Dynamic Cache, which decouples sequence-level and batch-level cache operations and avoids expensive split-merge procedures on large KV tensors.

Implications for deferred inference execution. In addition to preemption, per-expert queuing opens up possibilities for researchers to investigate optimizations in deferred execution, adaptive load balancing, dynamic workload distribution, fault tolerance, multi-tenant environments, and beyond. While QLLM is centered on low-latency inference for mixed-priority workloads, its foundational architectures are applicable to future systems demanding enhanced flexibility in MoE inference. Nevertheless, the concepts of per-expert queuing and low-overhead state management are not confined to MoE models and may be applied to dense models.

4 Evaluation

This section offers an initial evaluation of QLLM's performance utilizing practical hardware configurations, with an emphasis on its impact on the latency of LS requests and the overall turnaround time. We address the following questions:

- **Q1:** How does QLLM comply with the Latency SLO?
- **Q2:** How does QLLM affect throughput?
- **Q3:** How does QLLM affect the turnaround time for BE and LS requests?

Experimental Setup. The evaluation was conducted on a bare-metal system equipped with an Nvidia A100 GPU (80 GB memory), dual-socket Intel Xeon Gold 6336Y CPUs, 256 GB DRAM, and PCIe 4.0 interconnect.

Models and Dataset. We use Mixtral 8×7B, a representative of MoE models. The model is executed with 4-bit quantization and FP16 precision. In this configuration, the model required approximately 22.93 GB of GPU memory. Experiments are conducted using the ShareGPT dataset [24].

Baseline System. To evaluate the performance of QLLM, we compare it against the HF TGI inference engine, which represents a widely used production engine in serving LLM models. Importantly, QLLM is built on top of the HF engine, ensuring that any performance gains can be attributed to the scheduling mechanism rather than unrelated system differences. We set the maximum batch size to 32.

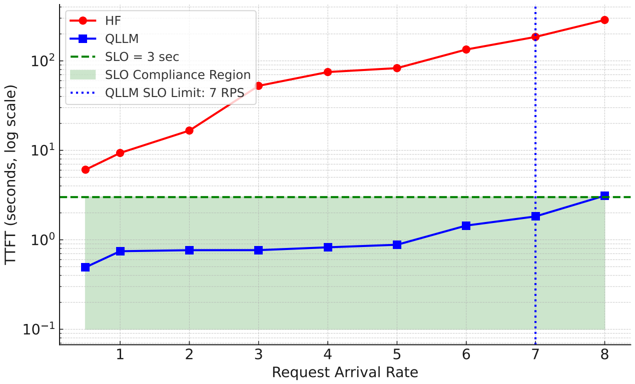


Figure 2: QLLM reduces TTFT for LS jobs by up to 101.6× while ensuring compliance with the SLO. In contrast, Hugging Face fails to meet the SLO even under low load due to priority-oblivious scheduling.

Workload. The workload generator retrieves prompts from the ShareGPT dataset and marks 20% of these prompts as LS prompts based on a random selection process. Subsequently, requests are dispatched to QLLM in accordance with Poisson arrival rates. It is important to know that QLLM performance may vary under different workload patterns, including the portion of LS requests and their distribution. However, in this preliminary evaluation of our prototype system, we explore the workload described earlier.

Evaluation Metrics. For users, the latency of LS requests significantly influences the responsiveness of applications like code completion and medical LLM applications. Therefore, we evaluate and present both TTFT and the turnaround time for LS jobs. Turnaround time refers to the complete delay from the moment a request is received by the system until the response is fully generated and delivered to the user. Furthermore, we measure the *job completion rate* as a throughput metric. Additionally, we investigate the impact of QLLM on the BE turnaround time.

Q1: QLLM significantly decreases TTFT latency. Figure 2 compares the TTFT of the HF TGI baseline (red line) with QLLM (blue line). We set the SLO to 3 seconds (green dashed line), which is 10× the processing iteration time of a single decode [21, 32]. The results show that the priority-oblivious scheduler of HF TGI cannot comply with the SLO even at low request rates. In fact, HF TGI employs a run-to-completion strategy in which lengthy BE jobs postpone the execution of LS jobs. In contrast, QLLM handles up to 7 jobs/s while still adhering to the SLO latency for LS requests. Notably, QLLM reduces TTFT by up to 101.6× and an average of 65.2× while complying with the SLO.

Q2: QLLM preserves overall throughput. Figure 3 depicts the variation in throughput as a function of the request arrival rates,

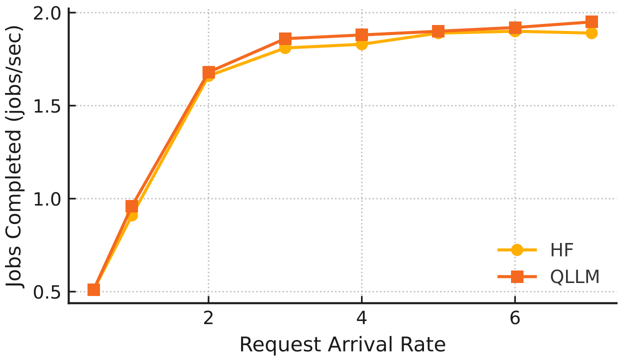
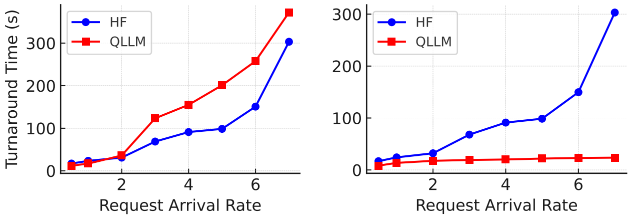


Figure 3: QLLM maintains a comparable or slightly higher job completion rate compared to HF.



(a) BE Requests **(b) LS Requests**

Figure 4: Comparison of turnaround times for Best Effort and Latency-Sensitive requests.

quantified by the job completion metric. QLLM demonstrates throughput that is comparable to, or slightly exceeds, the baseline while adhering to the latency SLO for LS requests. Its ability to execute fine-grained preemption enables QLLM to promptly address incoming LS requests while simultaneously managing best-effort decoding jobs, thereby increasing GPU efficiency. The throughput measured in tokens per second exhibits a similar trend.

Q3: QLLM reduces turnaround time for LS jobs. Figure 4 compares the turnaround time of HF TGI (blue line) and QLLM (red line) for BE and LS jobs. For LS jobs, QLLM reduces up to 12.8× the turnaround time relative to the baseline thanks to its preemption and low-latency response to LS requests. QLLM experiences a 1.38× increase in response time for BE requests, reaching a peak of 2.04×. The findings suggest that the QLLM advantages for LS jobs considerably surpass the detriments to BE requests.

5 Related Work

Existing approaches to LLM inference scheduling primarily optimize at the iteration level, lacking finer-grained control. ORCA [10] and FastServe [32] optimize scheduling at the iteration level, primarily benefiting dense models but lacking finer-grained control. Although FastServe introduces token-level preemption, it does not incorporate priority-aware execution. Additionally, Reef [9] focuses on microsecond-scale preemption for traditional DNN model serving, where each request involves a single inference pass through the model. In contrast, LLM inference operates autoregressively, requiring multiple iterations.

Llumnix [28] and FastSwitch [26] enhance scheduling by managing KV cache migration and preemptive context switching, respectively, but still operate within iteration-level scheduling. Unlike these approaches, our work achieves more fine-grained control, enabling efficient priority management and dynamic execution without excessive recomputation overhead.

In contrast, QLLM enables expert-level preemption and priority-aware scheduling, addressing head-of-line blocking and priority inversion. This fine-grained control ensures low latency for LS tasks while maintaining high throughput, surpassing prior iteration-level approaches.

6 Discussion

Limitations. While the existing QLLM prototype demonstrates promising results, we are actively exploring methods to reduce memory overhead and mitigate potential starvation in certain workloads. Compared to iteration-level preemption, our approach requires caching additional states (*e.g.*, *routing_weights* and *hidden_states*), though the primary memory constraint remains the KV cache. Efficient memory management to further enhancing the effectiveness of preemptive scheduling could be a subject for future work.

Overlapping preemption with execution. The new MoE layer introduces execution flexibility, allowing the system to process entire batches, selectively execute specific tokens or experts, or dynamically preempt ongoing tasks. This adaptability enables opportunities for overlapping memory operations with concurrent task execution, which can improve overall performance. Exploring these optimizations can further enhance system efficiency.

Applicability to dense models. Our approach extends beyond MoE models and is applicable to all LLM architectures. In dense models, preemptive scheduling at the layer level is more straightforward due to their deterministic execution, where all tokens follow the same computational path. However, MoE models introduce dynamic token-to-expert routing, necessitating more sophisticated state management and preemption mechanisms. By addressing these complexities, QLLM provides a generalized scheduling framework that supports both MoE and dense models.

QLLM's architectural flexibility introduces new opportunities for optimizing LLM inference, enabling more efficient scheduling and execution strategies. This design opens pathways for further exploration in adaptive workload management, memory-efficient preemption, and broader applications beyond MoE models. By redefining how inference tasks are scheduled and executed, QLLM lays the groundwork for future advancements in efficient and scalable LLM serving.

7 Conclusion

This paper introduces QLLM, the first inference system that facilitates fine-grained preemption and priority-aware scheduling for MoE models, optimizing latency-sensitive jobs while preserving high throughput. By implementing per-expert queues and a priority-aware scheduler, QLLM addresses HOL blocking and priority inversion, achieving a reduction in LS jobs TTFT latency by up to 101.6× and ensuring SLO compliance up to 7 requests/sec, whereas the baseline never adheres to SLO in tested scenarios. The proposed approach is modular and extensible, allowing seamless integration with existing MoE frameworks. Future research will focus on further optimizing memory consumption, potential starvation, and open-source QLLM to drive continued innovation in efficient LLM inference.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions on this paper. This work has been partially supported by Vinnova (the Sweden's Innovation Agency), the Swedish Research Council (agreement No. 2021-04212), KTH Digital Futures, and Knut and Alice Wallenberg Foundation (Wallenberg Scholar Grant for Prof. Dejan Kostić).

References

[1] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V Le. 2020. Towards a Human-like Open-Domain Chatbot. *arXiv preprint arXiv:2001.09977* (2020). https://arxiv.org/abs/2001.09977

[2] 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 *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024)*. Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[3] Vincent-Pierre Berges, Barlas Oğuz, Daniel Haziza, Wen-tau Yih, Luke Zettlemoyer, and Gargi Ghosh. 2024. Memory Layers at Scale. *arXiv preprint arXiv:2412.09764* (2024). https://arxiv.org/abs/2412.09764 Accessed: 2025-02-11.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). https://arxiv.org/abs/2107.03374

[5] Zhiyuan Dai et al. 2024. DeepSeek-MoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. *arXiv preprint arXiv:2406.00023* (2024).

[6] DeepSeek-AI et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. (2025). The DeepSeek-R1 model contains 671 billion parameters..

[7] Hugging Face. 2023. Text Generation Inference. https://github.com/huggingface/text-generation-inference. Accessed: 2025-02-06.

[8] Martin Iglesias Goyanes. 2024. LLM Inference at Scale with TGI. *Hugging Face* (2024). https://huggingface.co/blog/martinigoyanes/llm-inference-at-scale-with-tgi

[9] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[10] Minsoo Jeon et al. 2023. ORCA: A Fine-Grained Execution Model for Large Language Model Inference. *arXiv preprint arXiv:2301.10292* (2023). https://arxiv.org/abs/2301.10292

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

[12] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for microsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[13] Jakub Krajewski, Jan Ludziejewski, Kamil Adamczewski, Maciej Pióro, Michał Krutul, Szymon Antoniak, Kamil Ciebiera, Krystian Król, Tomasz Odrzygóźdź, Piotr Sankowski, Marek Cygan, and Sebastian Jaszczur. 2024. Scaling Laws for Fine-Grained Mixture of Experts. *arXiv preprint arXiv:2402.07871* (2024). https://arxiv.org/abs/2402.07871

[14] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[15] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 38–56. doi:10.1145/3651890.3672274

[16] Ramesh Nallapati, Bowen Zhou, Cicero dos Santos, Çağlar Gülçehre, and Bing Xiang. 2016. Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond. In *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*. 280–290. https://aclanthology.org/K16-1028/

[17] NVIDIA. 2023. FasterTransformer: A Fast Inference Toolkit for Transformer Based Models. https://github.com/NVIDIA/FasterTransformer.

[18] NVIDIA Corporation. 2023. NVIDIA Grace Hopper Superchip Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/ Accessed: 2025-01-14.

[19] NVIDIA Corporation. 2023. NVLink & NVSwitch: Fastest HPC Data Center Platform. https://www.nvidia.com/en-us/data-center/nvlink/ Accessed: 2025-01-14.

[20] Romain Paulus, Caiming Xiong, and Richard Socher. 2018. A Deep Reinforced Model for Abstractive Summarization. In *Proceedings of the 6th International Conference on Learning Representations*. https://arxiv.org/abs/1705.04304

[21] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 325–341. doi:10.1145/3132747.3132780

[22] Stephen Roller, Emily Dinan, Naman Goyal, Da Ju, Mary Williamson, Yinhan Liu, Jing Xu, Myle Ott, Eric Michael Smith, Y-Lan Boureau, and Jason Weston. 2021. Recipes for Building an Open-Domain Chatbot. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. 300–325. https://aclanthology.org/2021.eacl-main.24

[23] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1073–1083. https://aclanthology.org/P17-1099/

[24] ShareGPT Team. 2023. ShareGPT. https://sharegpt.com/. Accessed: 2025-01-19.

[25] Noam Shazeer et al. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *arXiv preprint arXiv:1701.06538* (2017).

[26] Ao Shen, Zhiyao Li, and Mingyu Gao. 2024. FastSwitch: Optimizing Context Switching Efficiency in Fairness-aware Large Language Model Serving. *arXiv preprint arXiv:2411.18424* (2024). https://arxiv.org/abs/2411.18424

[27] Diyi Shi, Hao Zheng, Hongyu Ren Zhang, et al. 2023. vLLM: A High-Throughput and Memory-Efficient Inference Engine for Large Language Models. *arXiv preprint arXiv:2305.11342* (2023). https://arxiv.org/abs/2305.11342

[28] 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)*. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[30] vLLM Project. 2024. vLLM's V1 Engine Architecture. https://github.com/vllm-project/vllm/issues/8779. Accessed: 2025-02-09.

[31] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. *arXiv preprint arXiv:2305.05920* (2023).

[32] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. FastServe: Fast Distributed Inference Serving for Large Language Models. *arXiv preprint arXiv:2305.05920* (2023). https://arxiv.org/abs/2305.05920

[33] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin