Supporting Multiple QoS Classes in LLM Inferencing Service

Anubhav Jana, Felix George, Uma Devi
IBM Research
India

Abstract

Large Language Models (LLMs) are increasingly popular for tasks such as text generation, code creation, and conversational AI due to their advanced natural language processing capabilities. These models heavily rely on GPUs for their parallel processing capabilities, crucial for handling the complex computations involved in inferencing tasks. Furthermore, different kinds of users have different latency requirements and common approaches tend to over-allocate resources to meet latency requirements. The high cost of GPUs prohibits dedicated GPUs per user. These inefficiencies result in higher operational costs and sub-optimal performance, especially when trying to meet diverse SLO (Service Level Objectives) requirements across different user classes. To address these challenges, we introduce a novel approach that utilizes statistical multiplexing of requests among user classes defined by their QoS (Quality of Service) requirements. Our methodology involves extensive benchmarking of GPUs across various profiles and LLM models. We use this benchmark to identify the request rate saturation point and establish latency bounds for certain percentiles (e.g. 50th, 80th, 95th) corresponding to the saturation request rate of the benchmarking inferencing instance. Using these benchmarks and the user's SLO for latency, we apply ILP (Integer Linear Programming) with a static priority-based multi-instance router to determine the optimal instance configurations and the percentage of request distributions to be sent to the instances. By incorporating priority classes, we demonstrate a significant reduction in latency and waiting time for higher-priority requests, achieving superior performance compared to configurations without prioritization. Our results highlight the importance of priority-based routing in maintaining SLOs while optimizing resource allocation and cost.

Introduction

Large Language Models (LLMs) have gained prominence for their capabilities in tasks such as text generation, programming code creation, chatbots, information extraction, and language translation.

LLM inferencing involves two primary steps: decoding and prefill. Decoding generates output tokens based on the input sequence, while prefill prepares the input data for efficient processing. Two notable inference serving systems are TensorFlow Serving (TGIS) and vLLM. TGIS provides scalable model deployment for production environments, and vLLM is optimized for LLMs, offering efficient memory management and high throughput. Scalability challenges, particularly in hardware requirements for both training and inference, necessitate optimizing LLM inferencing to enhance performance, drive cost-efficiency, and maximize hardware utilization. Efficient GPU management directly translates to operational cost savings by reducing power consumption and extending hardware lifespan. GPUs are costly, making dedicated GPUs for each user class prohibitive for many organizations striving to meet Service Level Objective (SLO) requirements. LLM services must support multiple users with varying demands for service (requests per minute, tokens per request) and SLOs for multiple LLM models. Given the high cost of GPUs, LLM services should – (I) support various users with minimal resources (number of GPUs, GPU cost), (ii) minimize energy consumption and (iii) optionally support a best-effort class that can utilize spare capacity.

An example of an inference serving pipeline is given in Figure 1. Inference requests are routed to the inference servers via a router. Upon arrival, requests are enqueued in the queue and based on the completion of some ongoing request, a new request gets admitted to the batch (The number of concurrent requests running on the GPU determines the batch size). Requests are picked up from the batch and processed to generate output tokens (*decode* phase) in an auto-regressive manner. On completion of a request, whenever a new request is admitted to the batch from the queue, the *prefill* phase starts for that new request (populating KV cache memory) before it goes into the decode phase.

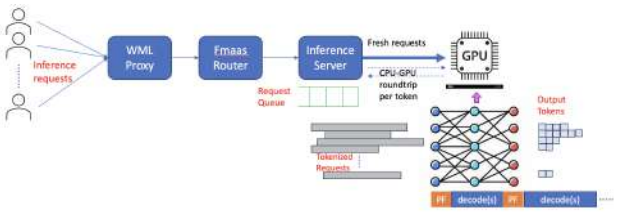


Figure 1: Inference Serving Pipeline.

Continuous batching is crucial in this context. It involves one or more decode phases followed by a prefill phase, with the time taken by a decode phase dependent on batch size. Each decode phase generates one token per request in the batch. The mean time per token (TBOT) is given by:

$$\text{TBOT} = \frac{T_{\text{decode}} \times N_{\text{decode}} + T_{\text{prefill}} \times N_{\text{prefill}}}{N_{\text{decode}}}$$

For a given batch size, TBOT can vary due to the presence or absence of prefill.

Time to First Token (TTFT) is:

$$\text{TTFT} = \text{Waiting time in queue} + \text{one prefill time}$$

Time in queue depends on arrival rate and service rate. For the first request in queue, time in queue is the time taken for one of the requests under processing to complete. At steady state, the time between successive request completions (TBRC) is:

$$\text{TBRC} = \frac{\text{TBOT} \times \text{TPR}_{\text{avg}}}{B}$$

where TPR_{avg} is the average number of tokens per request and B is the max batch size. The latency of the n^{th} waiting request is $n \times$ TBRC. The bound on time to first token can be determined by computing the probability of k waiting requests, $k \geq 0$.

To comprehensively understand the relationship between these metrics, extensive realistic benchmarking is essential. This benchmarking provides latency distributions, allowing us to establish percentile latency bounds for given request rates. Using these distributions, we employ Integer Linear Programming (ILP) to determine the optimal number of instances required and the percentage of request rates per user priority class (prioritized by their latency requirements) to allocate to each instance, ultimately optimizing costs.

To this effort, we present a novel approach with the following contributions in this paper:

1. **Extensive GPU Benchmarking**: We benchmark various GPU profiles and LLMs to identify performance saturation points under different configurations and establish latency bounds for incoming request rates from latency distributions. This information is then used to determine the number of required instances and allocate request rates per user class for the chosen instances.
2. **Optimized Instance Allocation with ILP**: We propose an Integer Linear Programming (ILP) model to optimize (minimize) the number of required instances and allocate request rates to the chosen instances, thereby minimizing resource costs while ensuring latency bounds are met for various user classes.
3. **Novel Static Priority-Based Multi-Instance Routing**: We develop a static priority-based multi-instance router that leverages ILP results and benchmarks to route requests. This approach enhances performance and resource efficiency by directing requests based on their priority levels (latency SLOs).
4. **Interactive Dashboard Development**: We develop a dashboard to visualize benchmarking results and router performance, providing a user-friendly interface for analyzing system metrics and decision-making based on our proposed methodologies.

Background and Related Work

Key Metrics for LLM Serving

Evaluating LLM inference speed involves several key metrics:

1. **Time To First Token (TTFT):** Time taken for the model's first output token to generate after submitting the query prompt.
2. **Time Per Output Token (TPOT):** Time required to generate each output token. For instance, a TPOT of 100 ms per token equals 10 tokens per second.
3. **Latency:** Total time to generate a complete response, calculated as latency = TTFT + TPOT * (number of tokens).
4. **Throughput:** Number of output tokens generated per second across all users.

The goal is to minimize TTFT and TPOT while maximizing throughput.

Impact of Hardware on LLM Performance and Efficient Inference Hardware Utilization

LLM performance is largely determined by memory bandwidth due to the dominance of matrix-matrix multiplication operations. The speed of token generation is more reliant on loading model parameters from GPU memory to local caches than on computation itself. Thus, memory bandwidth is a key predictor of generation speed.

Efficient inference hardware utilization is essential to manage costs, especially with expensive GPUs. Shared inference services lower costs by combining multiple user workloads, filling gaps, and batching overlapping requests. For large models, achieving optimal cost/performance requires large batch sizes, necessitating larger KV caches and more GPUs. Service operators must balance these trade-offs and optimize systems for efficient operation.

Related Work

Memory Management Managing KV cache memory is crucial for large language model (LLM) serving, especially with long context lengths. Predicting the generation length of requests is challenging, leading to inefficient pre-allocation of KV cache space. Earlier methods [NVIDIA2017] allocated based on a preset maximum request length, causing storage wastage when requests ended early.

To address this, S^3 [Jin et al.2023] predicted an upper bound for generation length, reducing waste but struggling with fragmented storage. vLLM [Lin et al.2024a] improved this by using a paged storage method, dividing large memory spaces into blocks and dynamically mapping KV cache non-contiguously, reducing fragmentation and improving throughput. LightLLM [ModelTC2024] further refined this

by allocating KV cache at the token level, reducing waste further.

Modern systems use paged storage to minimize redundant KV cache memory waste, but this introduces irregular memory access patterns for the attention operator. Efficient management of virtual and physical KV cache addresses is essential. For instance, vLLM's PagedAttention [Lin et al.2024a] stores the head size dimension as a 16-byte contiguous vector for the K cache, while FlashInfer [Ye2024] implements various data layouts and memory access schemes. Optimizing the attention operator in conjunction with paged KV cache storage remains a key challenge for enhancing LLM serving systems.

Continuous Batching In scenarios where request lengths within a batch differ, shorter requests may finish sooner, leading to inefficient utilization. Continuous batching, introduced by ORCA [Yu et al.2022], mitigates this by batching new requests as soon as old ones finish, enhancing efficiency. ORCA proposed iteration-level batching for LLM serving, encompassing pre-filling and decoding steps. vLLM [Lin et al.2024a] extended this to attention computation, enabling batching of requests with different KV cache lengths.

Further advancements include the split-and-fuse method by Sarathi [Agrawal et al.2023], DeepSpeed-FastGen [Holmes et al.2024], and Sarathi-Serve [Agrawal et al.2024], which splits long pre-filling requests and batches them with shorter decoding requests. LightLLM [ModelTC2024] also employs this technique, reducing tail latency by balancing workloads and preventing stalls. This approach leverages the auto-regressive nature of LLMs, maintaining mathematical equivalence and significantly reducing request latency

Scheduling Strategy In the context of serving LLMs, the variability in job lengths significantly impacts the throughput of the system, with the sequence in which requests are processed playing a crucial role. Head-of-line blocking [Wu et al.2023] occurs when long requests are prioritized, leading to rapid memory usage increase and subsequent requests being blocked once system memory is full.

ORCA [Yu et al.2022] was among the first systems to address this issue, with open-source systems like vLLM [Lin et al.2024a] and LightLLM [ModelTC2024] also employing a first-come-first-serve (FCFS) scheduling strategy for requests. DeepSpeed-FastGen [Holmes et al.2024], on the other hand, prioritizes decoding requests to enhance performance.

FastServe [Wu et al.2023] introduces a preemptive scheduling strategy to mitigate head-of-line blocking, achieving low job completion time (JCT) in LLM serving. It uses a multi-level feedback queue (MLFQ) to prioritize requests with the shortest remaining time. Due to the auto-regressive decoding approach which leads to unknown request lengths, FastServe predicts the length first and then uses a skip-join method to assign the appropriate priority to each request. VTC [Sheng et al.2024], unlike previous work, emphasizes fairness in LLM serving, introducing a cost function based on token numbers to ensure fairness among clients, and proposes a fair scheduler to uphold this fairness.

To achieve high throughput, LLM services are often deployed on distributed platforms. Recent research has aimed at optimizing the performance of such inference services by leveraging distributed characteristics. Splitwise [Patel et al.2023], TetriInfer [Hu et al.2024], and DistServe [Zhong et al.2024] have demonstrated the efficiency of separating the compute-intensive prefilling stage from the memory-intensive decoding stage, processing them independently based on their characteristics. SpotServe [Miao et al.2023] is designed to provide LLM services on clouds with preemptible GPU instances. It efficiently manages dynamic parallel control, and instance migration, and utilizes the auto-regressive nature of LLMs for token-level state recovery. Additionally, Infinite-LLM [Lin et al.2024b] extends the paged KV cache method used in vLLM to a distributed cloud environment.

Drawbacks Of the State-Of-the-Art Systems

1. None of the state-of-the-art systems perform extensive benchmarking across various GPU profiles and LLM models to leverage realistic latency distributions. These benchmarks are crucial for determining latency bounds at different request rates and deciding the optimal distribution of requests based on the latency bounds for saturating request rate across different inferencing instance profiles.
2. Isolation among users is often missing, leading to potential resource contention and performance degradation.
3. Meeting Service Level Objectives (SLOs) can require dedicated instances, which is not resource-efficient and is costly. Common approaches tend to over-allocate resources for meeting SLO requirements.
4. Simulating users is not realistic because:
   (a) There are concurrent users in many simulations, which do not accurately reflect real-world scenarios.
   (b) Our approach assumes a Poisson distribution for arrival rates for each user class, which better represents realistic workload arrival rates. This is because:
      i. Poisson distribution is widely used in modelling random events that occur independently over time, such as user requests in a network.
      ii. It captures the variability and burstiness of real-world traffic patterns more accurately than simplistic, simulation models.

System Design

Motivation

We present a motivating example to illustrate the problem and demonstrate how our the solution minimizes resources and costs while maintaining SLOs. Consider the following scenario: three users with Poisson-distributed request rates and specific latency requirements (lower latency requirement corresponds to higher priority).

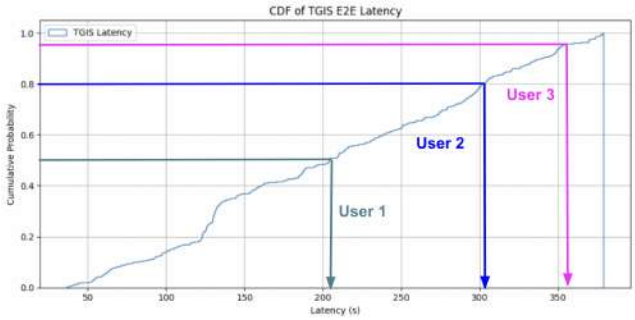


Figure 2: CDF of Latency Distribution on Llama A100 80 GB GPU operating at 15 RPM (Requests Per Minute).

1. **User 1**
   - Request Rate: 15 RPM
   - Latency: 204 s
2. **User 2**
   - Request Rate: 18 RPM
   - Latency: 301 s
3. **User 3**
   - Request Rate: 9 RPM
   - Latency: 352 s

SLO for User 1 cannot be met in an instance serving at 15 RPM (since only 50 percentile requests satisfies latency of around 204 seconds). It will require at least two dedicated instances. Similarly, the workload of User 2 cannot be satisfied with one instance per user since the input request rate (18 RPM) is greater than the GPU's operating request rate (15 RPM). However, the SLO of User 3 can be met with a single instance since its input request rate (9 RPM) is less than 15 RPM and can meet the desired latency. Simple FCFS scheduling will **require at least 5 inferencing instances** as given in Figure 3.

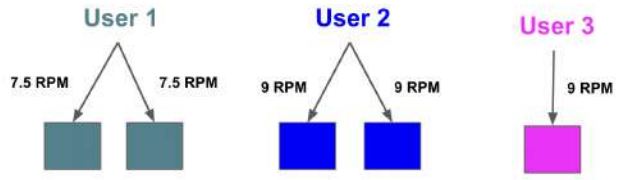


Figure 3: Meeting the SLO with 5 instances.

However, Figure 4 demonstrates how we can minimize the number of required instances to 3 by multiplexing the user requests instead of having 5 dedicated instances as in the case of Figure 3. Note that the multiplexing is statistical and is determined by ILP at runtime. Note that, the prioritization followed here is: **User 1 > User 2 > User 3 at each instance**. Thus, intelligent routing and simple priority scheduling can help meet all SLOs with just three instances.

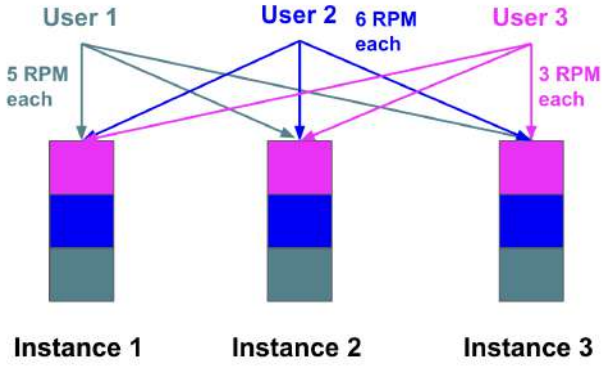


Figure 4: Meeting the SLO with just 3 instances.

Our Approach

Our basic intuition is to increase utilization by combining users with different latency requirements. We prioritize requests based on their latencies to optimize resource allocation and performance. Other priority policies can also be included, such as Earliest Deadline First (EDF).

We have the following possible approaches for providing QoS guarantees (per LLM Model).

- **Approach 1:**
  1. Identify the most efficient GPU and request configuration (e.g., A100 6 RPM) that can meet the SLO for each user class of a model. Instantiate one or more instances of the LLM service with dedicated GPUs for each instance for the class.
  2. **Pros:**
  (a) Simple to instantiate and implement.
  3. **Cons:**
  (a) Not cost or energy-efficient.
  (b) Scaling with changing load is not simple.

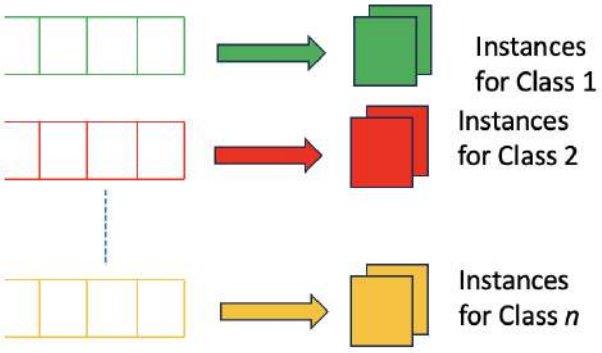


Figure 5: Approach 1: Separate instances for each user class.

- **Approach 2:**
  1. Multiple requests from multiple classes in a single instance while meeting their SLOs.

2. Distribute the requests of multiple classes to a model among one or more instances.
3. **Pros:**

(a) More cost and energy efficient.

(b) Adapting to changing loads from classes is simpler.

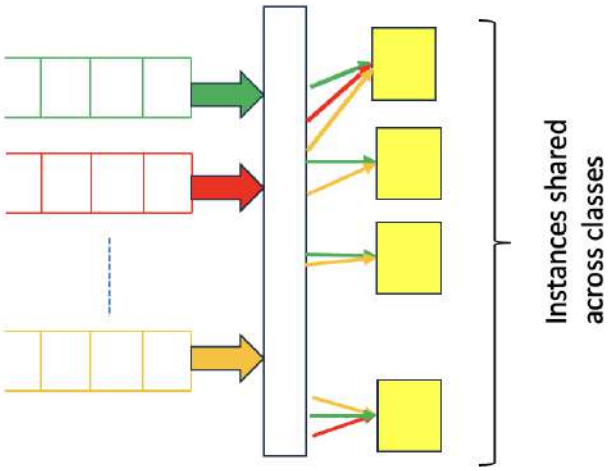


Figure 6: Approach 2: Multiplexing various user class requests across instances.

Figure 5 and Figure 6 illustrate the two approaches respectively.

Service Configuration - Problem Formulation

1. Given:

(a) M users, $U_1, U_2, \ldots, U_M$ with:

i. Request (query) rates $Q_1, Q_2, \ldots, Q_M$

ii. Latency bounds $L_1, L_2, \ldots, L_M$

(b) G GPUs, each of which can operate in:

i. R modes (request/service rates)

ii. F different frequencies

(c) Each GPU mode and frequency combination is associated with a latency function that maps a request rate $r \leq R$ to a latency bound (average, tail, or any percentile latency).

2. Objective:

(a) Determine how much rate to assign to each user on the different GPUs.

(b) Determine the mode and frequency at which each GPU has to be operated.

(c) Minimize power consumption while meeting the latency bounds of the user classes.

3. Note: This approach can be extended to include MIG instances too.

ILP Formulation

The Integer Linear Programming (ILP) model in this work optimizes the number of instances required and request rate allocations across multiple user classes. Its primary goal is to minimize active instances, thereby reducing operational costs and energy consumption while determining optimal rate allocations for each user class at specific instances operating under certain modes.

Notation

- $G_{i,j}$: Binary variables indicating GPU i operated in mode j, $1 \leq i \leq N$, $1 \leq j \leq R$.
- $A_{i,j}$: Aggregate rate for GPU i in mode j.
- $U_1, U_2, \ldots, U_M$: M user classes with:
  - Request (query) rates $Q_1, Q_2, \ldots, Q_M$.
  - Latency bounds $L_1, L_2, \ldots, L_M$.
- $X_{u,i,j}$: Decision variables denoting the rate allocation to user U_u on GPU i in mode j.
- $f_{i,j}(r)$: Denotes the latency in GPU i mode j for rate r.

Model Formulation

$$\text{Minimize} \quad \sum_{i=1}^{N} \sum_{j=1}^{R} G_{i,j}$$

$$\text{Subject to:} \quad \sum_{j=1}^{R} G_{i,j} \leq 1, \quad \forall i$$

$$\sum_{i,j} X_{u,i,j} = Q_u, \quad \forall u$$

$$\sum_{u,j} X_{u,i,j} \leq \sum_{j} G_{i,j} \cdot A_{i,j}, \quad \forall i$$

$$f_{i,j}\left(\sum_{k=1}^{u} X_{k,i,j}\right) \leq L_u, \quad \forall i, j, u$$

$$G_{i,j} \in \{0, 1\}, \quad \forall i, j$$

Note that one of the key challenges is establishing a relationship between request rate and latency to develop an equation that can properly determines latency as a function of the request rate.

Multi-Instance Router Architecture

Figure 7 illustrates the static priority-based multi-instance router architecture used for GPU resource allocation in our proposed approach.

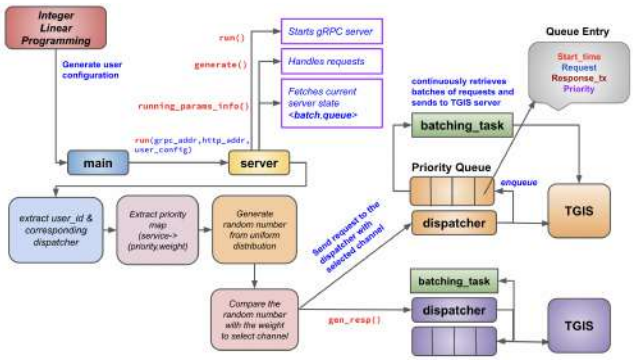


Figure 7: Static priority-based multi-instance router architecture for instance allocation.

Experiments

This section outlines the experiments conducted to evaluate the proposed system or model.

Benchmarking

We did extensive benchmarking across GPU profiles and multiple LLM models and the following were the profiles and their corresponding saturation points in Requests Per Minute (RPM) –

- Llama-8B A100 80GB (TGIS) : 13-15 RPM
- Llama-8B MIG 3g.40gb (TGIS) : 5-8 RPM
- Granite-3B A100 80GB (TGIS) : 55-60 RPM
- Bloom-760 A100 80GB (TGIS) : 135-140 RPM
- Bloom-760 MIG 3g.40gb (TGIS) : 65-70 RPM
- Granite-3B A100 80GB (vLLM)) : 280-300 RPM
- Llama-8B A100 80GB (vLLM) : 75-80 RPM

We deployed our custom load generator to dispatch requests to the inference servers, with arrival rates modelled by a **Poisson distribution**. Each benchmarking was performed for 40 minutes. Further, we discarded the first 8 minutes and the last 3 minutes of the experiment to get a stable state. This approach produced requests measured in ***Requests Per Minute (RPM)***. For each profile, we identified saturation points determined by memory bandwidth capacity, queue size, and batch size. Notably, after reaching the saturation point, the queue size continued to increase, whereas the batch size remained unchanged. Additionally, we recorded the mean and 95th percentile values for various metrics, including queue wait time, queue size, e2e latency, batch size, and the number of output tokens, across different request rates for each profile. To meet the SLOs for latency, we analyzed the latency distribution for each request rate, paying particular attention to the distribution at the saturation request rate. This analysis was crucial in determining the relationship between request rate and latency, which we formulated as a functional equation for use in the Integer Linear Programming (ILP) model.

Comparison between prioritization and non-prioritization

This experiment aimed to demonstrate the impact of priority on latencies and queue waiting times across different user classes. Conducted on an A100 80GB GPU using the Llama-8B model, we categorized users into three classes: Priority 1, 2, and 3, corresponding to users Alan, Noel, and Hari, respectively. Each user had a fixed request rate of 6 RPM, with a maximum of 60 output tokens generated per request. The experiment was performed in two iterations over a period of 40 minutes each – one with priority scheduling and one without.

User	Metric	95th Percentile	Mean
Alan	Latency (s)	**37.31**	31.21
	Queue Time (s)	8.46	3.27
Noel	Latency (s)	**102.27**	48.71
	Queue Time (s)	75.25	21.11
Hari	Latency (s)	**541.06**	223.43
	Queue Time (s)	511.48	195.78

Table 1: Performance metrics with priority

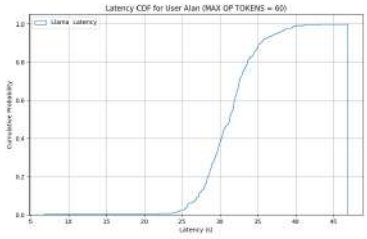
User	Metric	95th Percentile	Mean
Alan	Latency (s)	**364.95**	167.85
	Queue Time (s)	339.08	141.20
Noel	Latency (s)	**357.23**	170.57
	Queue Time (s)	336.12	143.71
Hari	Latency (s)	**361.03**	168.14
	Queue Time (s)	332.19	141.67

Table 2: Performance metrics without priority

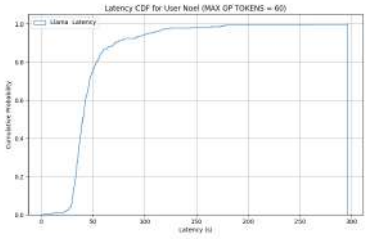
Table 1 clearly illustrates the significant reduction in both mean and 95th percentile latency and queue time for higher-priority users compared to lower-priority users. Conversely, Table 2 demonstrates that, without prioritization, latencies and queue times are almost uniform across all users. This contrast highlights that, with the appropriate instance configuration and intelligent routing, our approach can effectively meet the SLO requirements of QoS classes using minimal instances.

Figure 8 and Figure 9 present the cumulative distribution function (CDF) of latency distributions for different user priority classes under two distinct approaches: a priority-based approach and a non-priority-based approach. These figures illustrate the impact of each method on the end-to-end (e2e) latency experienced by users.

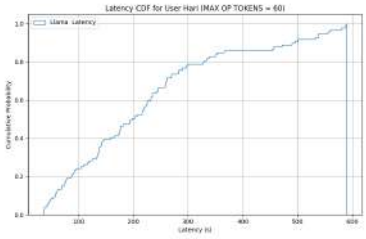
In the priority-based approach, the latency distribution demonstrated a clear differentiation between user classes. For Priority 1 user, the peak e2e latency was significantly reduced, reaching approximately 45 seconds. This indicated that high-priority users benefit from substantially lower latency. In contrast, Priority 2 user experienced a peak e2e latency of around 300 seconds, which was higher than that of Priority 1. However, Priority 3 user faced a much longer peak e2e latency of approximately 600 seconds, illustrating the trade-off in performance for lower-priority user classes.



(a) Alan's Latency CDF.

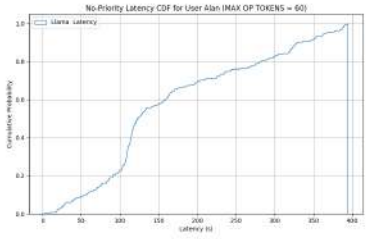


(b) Noel's Latency CDF.

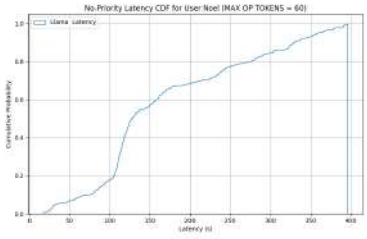


(c) Hari's Latency CDF.

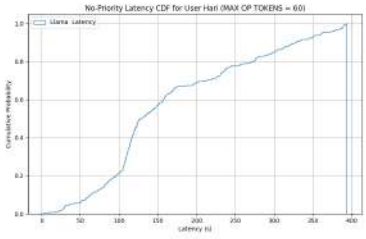
Figure 8: Priority Based Approach: CDF of E2E Latency.



(a) Alan's Latency CDF.



(b) Noel's Latency CDF.



(c) Hari's Latency CDF.

Figure 9: Without Priority: CDF of E2E Latency.

Conversely, the non-priority-based approach exhibited a different latency profile. Here, the peak e2e latency for all user classes converged to a uniform value of around 400 seconds. This uniformity highlighted the absence of prioritization, resulting in no distinct advantage for higher-priority user class over lower-priority ones. In this scenario, all users, regardless of their priority level, were subjected to similar latency experiences, thereby negating any preferential treatment of meeting SLO requirements.

The comparison between the two methods shows that the priority-based approach is better at reducing latency for high-priority users. By implementing prioritization, our approach ensures that critical requests are processed faster, thus enhancing the user experience for those deemed to have higher importance in terms of their SLO requirements. On the other hand, the non-priority-based approach, while simpler, fails to provide any latency benefits based on user priority, leading to a more homogeneous and less efficient distribution of processing time.

Priority-Based Multi-Instance Routing

The primary focus of our work is to develop an intelligent routing mechanism that directs user class requests to suitable multiple instances (decision obtained from ILP), ensuring compliance with Service Level Objectives (SLOs). To this effort, we employ a multi-instance routing strategy based on specific constraints and decision-making criteria to achieve this. The objective of this experiment was to assess the effectiveness of this multi-instance routing approach. The experiment was carried out in two setups (Priority of Alan $>$ Noel $>$ Hari) – (I) We sent one-third of the total requests of each user class to each of the three instances with total request rates of 10, 15 and 20 respectively. (ii) We sent all the requests to a single instance with total request rates of 4, 4 and 4 respectively for each user class.

Table 3 illustrates the distribution of requests for each user class across multiple instances running Llama. The mean latency and queue time metrics demonstrate that the priority levels are respected within each instance. Table 4 presents the performance when all requests are directed to a single instance. The results indicate that priority levels are still maintained. It is important to note that the Integer Linear Programming (ILP) model determines the actual rate allocation and assigns requests to specific instances operating at different request rates (modes).

Conclusion

In this paper, we presented a novel approach to optimize resource allocation and performance in LLM serving by leveraging extensive GPU benchmarking, Integer Linear Programming (ILP), and a static priority-based multi-instance router. Our methodology addresses the inefficiencies and high operational costs associated with over-allocating resources to meet diverse Service Level Objectives (SLOs) across different user classes.

Our extensive benchmarking of various GPU profiles and LLM models allowed us to identify performance saturation points and establish latency bounds for different percentiles across various request rates. Using this benchmark data, we applied ILP to optimize the number of required instances and allocate request rates per user class, thereby minimizing resource costs while ensuring latency bounds are met.

Table 3: Performance Statistics for Multiple Llama3 Instances (10:15:20)

User	Mean Queue Time	Mean Latency
Llama3-1		
Alan	79.89 ms	161.49 ms
Noel	148.88 ms	233.40 ms
Hari	207.15 ms	289.54 ms
Llama3-2		
Alan	85.57 ms	166.08 ms
Noel	137.06 ms	217.75 ms
Hari	205.60 ms	288.70 ms
Llama3-3		
Alan	87.04 ms	166.60 ms
Noel	135.56 ms	216.08 ms
Hari	208.88 ms	291.26 ms

Table 4: Performance Statistics for Single Llama3 Instance (4:4:4)

User	Mean Queue Time	Mean Latency	90th Percentile Latency
Alan	68.16 ms	134.10 ms	221.58 ms
Noel	95.79 ms	162.07 ms	257.25 ms
Hari	123.19 ms	190.50 ms	331.81 ms

The static priority-based multi-instance router we developed significantly enhances performance and resource efficiency by routing requests based on their priority levels (latency SLOs). Experimental results demonstrated a substantial reduction in latency and queue times for higher-priority requests compared to configurations without prioritization.

Overall, our approach demonstrates the effectiveness of combining extensive benchmarking, ILP optimization, and priority-based routing to meet diverse SLO requirements while optimizing resource allocation and costs in LLM serving. This methodology can be instrumental in improving the efficiency and performance of LLM services, particularly in scenarios with varying latency requirements across different user classes.

Future Work

This is still an ongoing work and we plan to integrate all the components and build and end-to-end system inference serving system. The system will handle varying request rates and latency bounds per user class. The ILP model will generate user configurations, determining the minimum number of instances, which instances to choose, and how to distribute user request rates across these instances at runtime. This configuration will be fed into a multi-instance router that prioritizes user class requests and sends them to the inference service for processing. Our goal is to demonstrate that, even with varying request rates and latency requirements, this end-to-end system can meet the SLOs for each user class using the minimal number of instances.

Appendices Following are the links to our code-base which are still an ongoing work.

- Load Generator: Load Generator Code.
- LLM Router: Multi-Instance Code.
- Benchmarking: Benchmarking Data and Utility Scripts
- ILP: ILP code
- Dashboard: Dashboard Code - Download the folder and run python3 dashboard.py.

Figure 10 displays a snapshot of our work-in-progress dashboard with metrics and distributions where instance type is chosen as **TGIS**, model as **Bloom-760** and GPU profile as **A100 80 GPU**

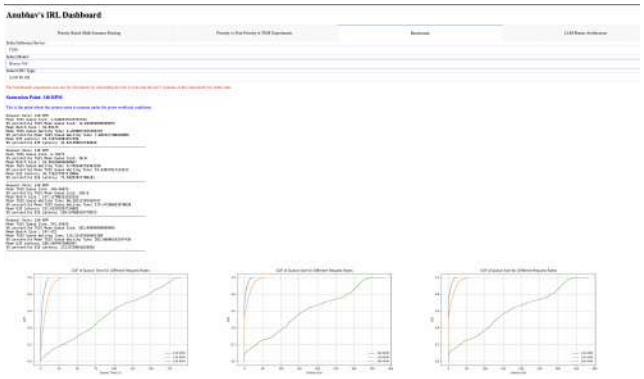


Figure 10: Dashboard snapshot displaying benchmark data for TGIS On Bloom-760 on A100 GPU.

References

[Agrawal et al.2023] Agrawal, A.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B.; and Ramjee, R. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint* arXiv:2308.16369.

[Agrawal et al.2024] Agrawal, A.; Kedia, N.; Panwar, A.; Mohan, J.; Kwatra, N.; Gulavani, B. S.; Tumanov, A.; and Ramjee, R. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint* arXiv:2403.02310.

[Holmes et al.2024] Holmes, C.; Tanaka, M.; Wyatt, M.; Awan, A. A.; Rasley, J.; Rajbhandari, S.; Aminabadi, R. Y.; Qin, H.; Bakhtiari, A.; Kurilenko, L.; and He, Y. 2024. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint* arXiv:2401.08671.

[Hu et al.2024] Hu, C.; Huang, H.; Xu, L.; Chen, X.; Xu, J.; Chen, S.; Feng, H.; Wang, C.; Wang, S.; Bao, Y.; Sun, N.; and Shan, Y. 2024. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint* arXiv:2401.11181.

[Jin et al.2023] Jin, Y.; Wu, C.-F.; Brooks, D.; and Wei, G.-Y. 2023. S3: Increasing gpu utilization during generative inference for higher throughput. *arXiv preprint* arXiv:2306.06000.

[Lin et al.2024a] Lin, B.; Peng, T.; Zhang, C.; Sun, M.; Li, L.; Zhao, H.; Xiao, W.; Xu, Q.; Qiu, X.; Li, S.; Ji, Z.; Li, Y.; and Lin, W. 2024a. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint* arXiv:2401.02669.

[Lin et al.2024b] Lin, B.; Peng, T.; Zhang, C.; Sun, M.; Li, L.; Zhao, H.; Xiao, W.; Xu, Q.; Qiu, X.; Li, S.; Ji, Z.; Li, Y.; and Lin, W. 2024b. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint* arXiv:2401.02669.

[Miao et al.2023] Miao, X.; Shi, C.; Duan, J.; Xi, X.; Lin, D.; Cui, B.; and Jia, Z. 2023. Spotserve: Serving generative large language models on preemptible instances. *arXiv preprint* arXiv:2311.15566.

[ModelTC2024] ModelTC. 2024. Lightllm. [Online], Available: https://github.com/ModelTC/lightllm.

[NVIDIA2017] NVIDIA. 2017. Fastertransformer: About transformer related optimization, including bert, gpt. [Online], Available: https://github.com/NVIDIA/FasterTransformer.

[Patel et al.2023] Patel, P.; Choukse, E.; Zhang, C.; Goiri, I.; Shah, A.; Maleki, S.; and Bianchini, R. 2023. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint* arXiv:2311.18677.

[Sheng et al.2024] Sheng, Y.; Cao, S.; Li, D.; Zhu, B.; Li, Z.; and Zhuo, D. 2024. Fairness in serving large language models. *arXiv preprint* arXiv:2401.00588.

[Wu et al.2023] Wu, B.; Zhong, Y.; Zhang, Z.; Huang, G.; Liu, X.; and Jin, X. 2023. Fast distributed inference serving for large language models. *arXiv preprint* arXiv:2305.05920.

[Ye2024] Ye, Z. 2024. flashinfer. [Online], Available: https://github.com/flashinfer-ai/flashinfer.

[Yu et al.2022] Yu, G.-I.; Jeong, J. S.; Kim, G.-W.; Kim, S.; and Chun, B.-G. 2022. Orca: A distributed serving system for transformer-based generative models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, 521–538.

[Zhong et al.2024] Zhong, Y.; Liu, S.; Chen, J.; Hu, J.; Zhu, Y.; Liu, X.; Jin, X.; and Zhang, H. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint* arXiv:2401.09670.