# Queue Management for SLO-Oriented Large Language Model Serving

Archit Patke
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
apatke@illinois.edu

Dhemath Reddy
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
dhemath2@illinois.edu

Saurabh Jha
IBM Research
Yorktown Heights, New York, USA
Saurabh.Jha@ibm.com

Haoran Qiu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
haoranq4@illinois.edu

Christian Pinto
IBM Research
Dublin, Ireland
Christian.Pinto@ibm.com

Chandra Narayanaswami
IBM Research
Yorktown Heights, New York, USA
chandras@us.ibm.com

Zbigniew Kalbarczyk
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
kalbarcz@illinois.edu

Ravishankar Iyer
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
rkiyer@illinois.edu

## ABSTRACT

Large language model (LLM) serving is becoming an increasingly critical workload for cloud providers. Existing LLM serving systems focus on interactive requests, such as chatbots and coding assistants, with tight latency SLO requirements. However, when such systems execute batch requests that have relaxed SLOs along with interactive requests, it leads to poor multiplexing and inefficient resource utilization. To address these challenges, we propose QLM, a queue management system for LLM serving. QLM maintains batch and interactive requests across different models and SLOs in a *request queue*. Optimal ordering of the request queue is critical to maintain SLOs while ensuring high resource utilization. To generate this optimal ordering, QLM uses a Request Waiting Time (RWT) Estimator that estimates the waiting times for requests in the request queue. These estimates are used by a global scheduler to orchestrate LLM Serving Operations (LSOs) such as request pulling, request eviction, load balancing, and model swapping. Evaluation on heterogeneous GPU devices and models with real-world LLM serving dataset shows that QLM improves SLO attainment by 40–90% and throughput by 20–400% while maintaining or improving device utilization compared to other state-of-the-art LLM serving systems. QLM's evaluation is based on the production requirements of a cloud provider. QLM is publicly available at https://www.github.com/QLM-project/QLM.

## CCS CONCEPTS

• **Computing methodologies → Machine learning**; **Distributed algorithms**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

## KEYWORDS

large language models, machine learning inference, queuing

**ACM Reference Format:**
Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2024. Queue Management for SLO-Oriented Large Language Model Serving. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3698038.3698523

## 1 INTRODUCTION

**Motivation.** Large language models (LLMs) such as OpenAI GPT-4 and Google Gemini have enabled novel capabilities in a wide range of AI applications [3, 50, 52] such as chatbots and coding assistants. These base models such as GPT4 are further fine-tuned to support specialized tasks such as copywriting, financial planning, etc. [33]. Consequently, serving multiple models for enterprise and consumer applications with latency-oriented service-level objectives (SLOs) has become increasingly critical [19, 35, 48].

Previous work in this area [16, 20, 23, 31, 36, 43, 49, 51] largely focused on serving interactive requests, such as chatbots, with tight latency SLO requirements. However, the recent explosive growth of LLM applications, has generated a need to support batch LLM queries with SLO values ranging from minutes to hours for tasks such as data wrangling [29], document processing [18], and model fine-tuning [44].

Given the broader range of SLO requirements and the use of multiple models, maintaining *request queues* is beneficial. Conceptually, when resources are limited, requests with relaxed SLOs can be kept at the back of the request queue and do not need to be executed immediately, while requests with tight SLOs can be kept ahead in the queue to ensure immediate execution and prevent *head-of-line (HOL) blocking*. Hence, queue ordering becomes an important decision-making problem for SLO-oriented LLM serving.

Previous work in SLO-oriented serving has largely focused on traditional DNN serving workloads such as CNNs and RNNs [15, 41], where the queue ordering decisions are made by systems such as Clockwork [13], INFaaS [39], and SHEPHERD [54]. Such systems leverage the deterministic execution times of DNN workloads to estimate queuing times and enable optimal ordering decisions that would maintain request SLOs. However, such systems cannot be easily applied to LLM serving workloads because the execution time per request is non-deterministic as the number of output tokens are unknown apriori. Using these system's assumption of fixed size batches with deterministic execution times for scheduling decisions leads to sub-optimal scenarios, described below.

First, previously proposed serving systems are unable to effectively multiplex interactive and batch requests on the same serving instance. We find that the queue waiting time in LLM serving is lower than estimate used by the scheduler in such systems as shown in Figure 1 (left). As these systems generate a higher than actual estimate of the total queuing time, they allocate batch and interactive requests on separate serving instances, even in scenarios where allocation on the same instance would suffice to meet SLOs.

Second, previously proposed systems are unable to effectively multiplex different models on the same serving instance. Similar to the first scenario, systems such as Clockwork [13] overestimate the time required to serve a single model and prefer to allocate independent instances for each model to preserve SLOs. Others such as SHEPHERD [54], altogether forgo serving multiple models on the same instance

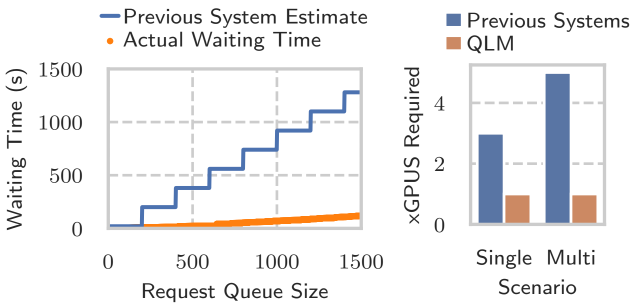

**Figure 1: Previously proposed SLO-oriented serving systems overestimate queue waiting time leading to suboptimal resource usage. (Left) Estimated waiting time when requests are run with Llama-70B on A100 GPUs with vLLM. (Right) Number of GPUs required to maintain 20s time-to-first-token (TTFT) SLO with previous systems vs QLM in single and multi-model scenarios.**

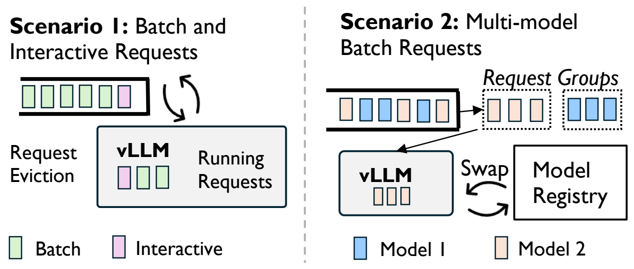

**Figure 2: QLM uses request groups and LLM Serving Operations (LSOs) such as request eviction to minimize resource requirement. Previously proposed systems would use four vLLM instances (compared to two for QLM) due to limitations described in Figure 1.**

due to the high cost of model swapping relative to inference times.

**Our Work.** To address these limitations, we propose QLM, a queue management system for LLM serving that maximizes SLO attainment while maintaining high throughput. At the core of QLM is the *Request Waiting Time (RWT) Estimator* (described in Section 6) that estimates the waiting times for requests in the request queue. We observe that as the queue size grows larger, statistical effects of continuous batching [51] in LLM serving allows us to create a tighter bound on waiting time. Additionally, we empirically validate and provide a proof for applicability of the RWT estimator. QLM leverages this improved estimation to allow for better utilization and decrease resource cost by making closer to optimal request queue reordering decisions.

To order requests in the queue, QLM, similar to SHEPHERD, uses the abstraction of *request groups* (described in Section 4). Each request group is a collection of requests that have relatively homogeneous performance requirements (e.g. similar SLOs and model). QLM places these request groups in *virtual queues* that determine the order in which requests are consumed by the LLM serving instances. In the LLM serving context, in addition to the benefits of scalability and predictability found in DNN serving, request groups are a useful abstraction for model swapping. As decision making is made at the per-request group level, the total amount of swapping is minimized and throughput is improved as depicted in Figure 2 (Scenario 2).

Finally, QLM uses a *Global Scheduler* (described in Section 7) that utilizes the waiting time estimates of request groups to create an optimal ordering and assignment of request groups onto the virtual queues to maximize SLO attainment. The virtual queue ordering is used by four basic LLM Serving Operations (LSOs) to manage the request queue (see Section 5 for details): (1) *Request Pulling* from the global waiting queue into the running batch in the GPU, (2) *Request Eviction* from the running batch back into the waiting queue, (3) *Model Swapping* from CPU to GPU memory, and (4) *Load Balancing* across multiple LLM model instances. For example, request eviction allows serving batch and interactive requests on the same LLM serving instance and prevents HOL blocking for interactive requests as shown in Figure 2 (Scenario 1).

**Results.** We demonstrate QLM on vLLM [20] as the backend LLM-serving system. We evaluate QLM on three popular LLMs of varying sizes (i.e., Mistral-7B [17], Vicuna-13B [5], and Llama-70B [47]) on GPU clusters with NVIDIA A10 and A100 GPUs. We adopt workloads from a real-world LLM dataset: ShareGPT [46] using setups derived from our production requirements. Our experiments demonstrate the following major improvements with QLM:

(1) *SLO Attainment:* Depending on the arrival rate, QLM achieves 40–90% higher SLO attainment compared to the vanilla vLLM serving system and 50–90% higher SLO attainment compared to traditional ML serving systems like SHEPHERD.

(2) *Request Throughput:* QLM improves the request throughput in a multi-model serving system by 400% on average and in a single-model serving system by 20% on average compared to other LLM serving systems.

(3) *LSO Ablation Study:* QLM demonstrates that all LSOs contribute to SLO attainment and throughput improvement. Notably, we find that model swapping improves throughput by 300% in multi-model serving, and request eviction improves SLO attainment by 80% in single-model serving.

QLM has been merged into an internal production LLM routing service.

## 2 BACKGROUND

### 2.1 LLM Inference

**Inference Primer.** An inference process starts from a request (prompt) with a list of input tokens $(x_1, \ldots, x_n)$. The LLM generates a list of output tokens $(x_{n+1}, \ldots, x_{n+T})$. Due to the *autoregressive* pattern, the LLM can only generate new tokens one by one, and the generation process of each new token depends on all the previous tokens in that sequence, specifically their key and value vectors. In this sequential generation process, the key and value vectors of existing tokens are cached for generating future tokens, known as *KV cache*.

Therefore, given a LLM request prompt, the generation computation can be decomposed into two phases: (1) A *prefill* stage takes the whole user prompt $(x_1, \ldots, x_n)$ as input and computes the probability of the first output token $P(x_{n+1}|x_1, \ldots, x_n)$. (2) A *decoding stage* (autoregressive generation) generates the remaining output tokens sequentially. At iteration $t$, the model takes one token $x_{n+t}$ as input and computes the probability $P(x_{n+t+1}|x_1, \ldots, x_{n+t})$ with the key vectors $k_1, \ldots, k_{n+t}$ and value vectors $v_1, \ldots, v_{n+t}$. This phase completes when an end-of-sequence (`<eos>`) token is emitted.

**Continuous Batching.** During LLM inference, the decoding stage is memory-bound, as loading model weights from memory takes longer than computation. Therefore, state-of-the-art LLM serving systems like vLLM [20], Orca [51], Tensor-RT [31] and TGI [16] employ continuous batching with iterative scheduling to enable dynamic addition of requests to a batch as soon as others have finished generation.

**PagedAttention.** Static allocation of the KV cache can result in significant memory waste as the KV cache grows dynamically during the decoding stage. PagedAttention [20] introduces the idea of managing the KV cache, like OS memory, via pages and enabling dynamic allocation. Such dynamic allocation prevents fragmentation and enables nearly 100% utilization of GPU memory and furthers throughput improvement when combined with continuous batching [20].

### 2.2 LLM Serving Systems

User-facing applications such as chatbots, log processing, and recommenders have specific latency SLO requirements [4, 19, 35, 37]. When interacting with LLMs, each application generates requests that consist of the *input prompts* and associated *metadata* (e.g., model type and SLO value) to the LLM serving system. For example, chatbots require requests to complete by a deadline (e.g., p99 time to first token (TTFT) < 20s [12]), and batch jobs like document processing have a more relaxed SLO in the order of minutes to hours. Requests may also need to be served by multiple fine-tuned models specialized for various tasks. For example, Code Llama [40]

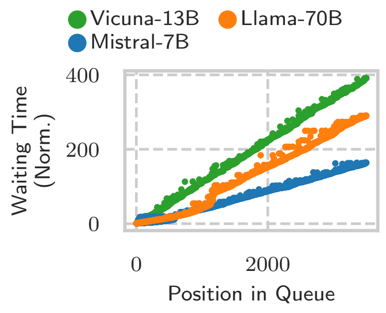

**Figure 3: Requests have predictable waiting times in a continuous batching system.**

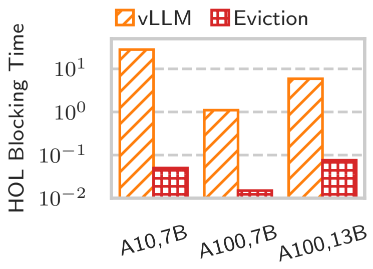

**Figure 4: Forced request eviction leads to reduction in head-of-line (HOL) blocking time.**

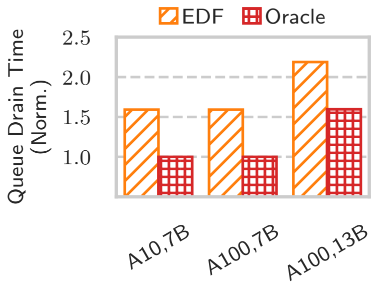

**Figure 5: Model swapping and request pulling can jointly decrease queue drain time.**

is fine-tuned for coding assistance, and Llama-chat [47] is fine-tuned for chatbots. Maintaining a standalone LLM serving system for each of these models or SLO types can be expensive, and they often have to be multiplexed together to share the same serving system [6, 21, 23, 42]. Multiplexing batch and interactive requests across various models with limited resources leads to formation of request queues, and managing these queues is critical to comply with SLO requirements.

## 2.3 QLM Definitions

Before we present the motivation and characterization study, below are definitions of the terms that we use.

**Definition 2.1.** *Request:* Each request consists of the prompt (i.e., input tokens) and its associated metadata (e.g., model type). Requests arrive at varying rates and burstiness, leading to request queues with dynamically changing sizes.

**Definition 2.2.** *SLO:* Each request arrives with a service-level objective (SLO) value that enforces the *time to first token (TTFT)* for the request. While QLM primarily focuses on TTFT, it can be paired with another system such as Andes [25] to also maintain inter-token latency (ITL).

**Definition 2.3.** *LLM Serving Instance:* An LLM serving system[1] is capable of hosting LLM models by providing the necessary infrastructure and resources to load the models into memory and respond to requests. QLM is compatible with existing LLM serving systems such as vLLM [20] and TGI [16]. An LLM serving instance is composed of the LLM serving system and an LLM model that is being served.

[1]The difference between an LLM serving system and an LLM serving instance is similar to JAVA classes and objects. For example, vLLM is an LLM serving system while vLLM with a loaded model like Llama 70B is an instance of the LLM serving system.

## 2.4 Motivation and Characterization

To enable SLO-oriented LLM serving, it is critical to understand (1) the impact of LLM autoregressive patterns on the request waiting time, (2) the performance implications of queuing batch and interactive requests, and (3) the performance implications of queuing multi-model batch requests.

We characterize these scenarios with a state-of-the-art LLM serving system, vLLM [20], to motivate the design of QLM. We use ShareGPT [46] traces to evaluate the system. We present three key insights below.

**Insight #1:** ***Waiting times in long request queues can be accurately estimated analytically.*** Recall from Figure 1 that incorrectly estimating waiting times can lead to resource waste. Therefore, we attempt to build an accurate queue waiting time estimator.

While the completion time for individual requests in LLM inference can vary widely, the average waiting time for a request in a long queue is predictable. Due to the statistical averaging effects resulting from a large number of requests (as detailed in Appendix A.1), the waiting time can be estimated by dividing the total number of output tokens for the pending requests by the token generation throughput. The total number of output tokens adheres to a Normal distribution, in accordance with the Central Limit Theorem, since individual requests are independent of each other

Figure 3 illustrates this linear relationship between waiting time and queue position when serving requests for three varying-sized LLMs on NVIDIA A100 GPUs. Additionally, we find that the estimator is highly accurate with a coefficient of determination ($R^2$) of 0.99 (out of 1.0).

**Insight #2:** ***HOL blocking times due to continuous batching can be in the order of tens of seconds.*** The straightforward way to prioritize interactive requests over batch requests on the same LLM serving instance is by placing them at the front of the waiting queue. However, request placement in the waiting queue may not be sufficient for immediate execution due to the lack of available GPU memory, causing head-of-line (HOL) blocking. Therefore, in such scenarios, evicting batch requests from the GPU is required. To minimize the cost of eviction, we can preserve the KV cache of batch requests, allowing execution to resume from the last decoding iteration.

Figure 4 illustrates the HOL blocking time when run with a mixed workload comprising interactive and batch requests. In the absence of request eviction, the HOL blocking time can be in the order of several seconds, which can lead to violation of latency SLOs for interactive requests. However, request eviction significantly reduces the waiting time because interactive requests only need to wait for a single decoding iteration before they can be scheduled, resulting in a 100−1000× reduction in waiting time.

**Insight #3:** ***Policies such as Earliest Deadline First (EDF) are insufficient to eliminate HOL blocking from model swapping.*** The optimal request pulling strategy to maximize the number of requests that satisfy SLOs is Earliest Deadline First (EDF) scheduling. *However, this assumes that the model swapping cost is negligible.* Frequent model swaps can happen (similar to thrashing) if multiple models are served to time share the same GPU devices, leading to SLO violations due to longer completion times to drain the queue and a drop in throughput. For example, consider the case illustrated in Figure 5. Requests with varying SLOs arrive in the queue, and they are placed by an EDF policy, causing multiple model swaps and substantially higher time to drain the entire request queue. Specifically, we find that across models and GPUs, the time required to serve all requests in the queue (i.e., the queue drain time) is substantially higher for the EDF policy compared to an Oracle policy that groups requests from the same model together to prevent the overhead of repetitive model swaps.

## 3 QLM DESIGN OVERVIEW

QLM aims to maximize latency SLO satisfaction in LLM serving workloads. To do so, QLM manages a global request queue and orchestrates multiple LLM Serving Operations (LSOs) to reorder and drain the global queue.

### 3.1 Lifecycle of a Request in QLM

To explain the design of QLM (as shown in Figure 6), we first walk through the lifecycle of a request generated from applications to completion. LLM-augmented applications generate requests that are received at the QLM API gateway. These requests are added to a global queue where they wait until being served. To prevent the global queue from being a single point of failure, it is implemented with a distributed message broker such as RabbitMQ [38] that provides the requisite fault tolerance and consistency properties.

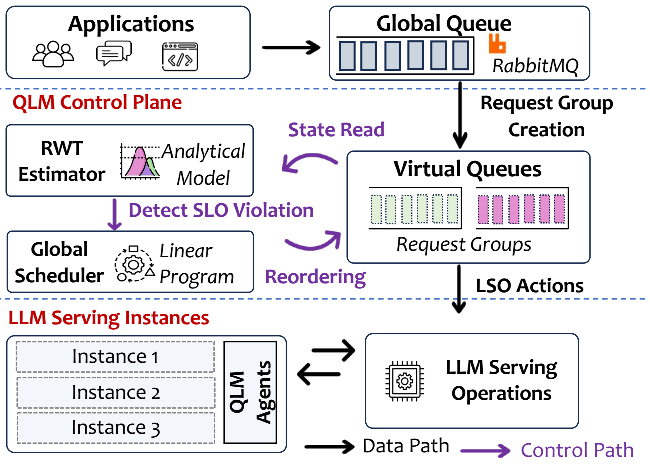

**Figure 6: Overview of QLM.**

**Formation of Request Groups.** Every incoming request is grouped with other requests that share common performance characteristics (such as model type, SLO value, and token distribution) to form *Request Groups*. This converts the complexity of the optimization problem from per-request level to per-request-group level. By doing so, it alleviates the scalability challenges and lowers optimization overheads. Additionally, request groups are a useful abstraction in the multi-model serving case as described in Section 2.4. Request grouping criteria and details are described in Section 4.

**Assigning Request Groups to Virtual Queues.** Requests in a request group are then assigned to a *Virtual Queue*, representing a waiting queue for an LLM serving instance in the cluster. The introduction of virtual queues creates a common abstraction for setting the actions of backend LLM Serving Operations (LSOs) such as request pulling, request eviction, load balancing, and model swapping. The ordering of the request groups in a virtual queue determines the execution ordering of the requests on the corresponding LLM serving instance. We refer readers to Section 4 for virtual queue formation and Section 5 for the translation from virtual queue ordering to LSO actions.

**Virtual Queue Reordering for SLO Attainment Maximization.** While requests are assigned to request groups in a first-come-first-serve manner, request groups in a virtual queue are reordered to maximize the SLO attainment for all requests being served. At the core of SLO attainment maximization are QLM's *request waiting time (RWT) estimator* (see Section 6) and *global scheduler* (see Section 7).

**Request Execution.** Each request, when being moved to the head of the virtual queue, will be executed on the LLM serving instance, and the output will be returned to the application. This completes the lifecycle of a request.

We illustrate the rationale of QLM's design choices and workflow in the next section (Section 3.2).

## 3.2 QLM Design Principles

We highlight the major design principles underpinning QLM, derived from large-scale production LLM serving workload requirements at a major cloud provider.

**Design Principle #1:** ***Scaling to a high request arrival rate and burstiness.*** QLM must be able to handle a high volume of requests for SLO attainment without the overhead that compromises the serving throughput. Existing model-serving frameworks that leverage optimization techniques such as linear programming have exponential or cubical complexity, which limits the scalability to larger workloads and longer queues. QLM instead introduces *request groups* to reduce the input space of the optimization solvers, thus lowering the computational overhead and enabling scalability.

**Design Principle #2:** ***Handling multiple LSOs with inter dependencies.*** To attain model-serving latency SLOs, it is critical to translate the latency SLO to the appropriate backend LSO actions. QLM models these complex interrelationships with a two-step approach. First, *virtual queues* enable the necessary abstraction to enable actions for multiple backend LSOs. Second, the *global scheduler* models the impact of ordering on multiple LSOs with a linear programming solver. We specifically prefer a linear programming solver over other optimization methods because it systematically considers various constraints introduced by multiple LSOs, SLO constraints, and waiting time estimates from the RWT estimator.

**Design Principle #3:** ***Handling heterogeneous models and hardware device configurations.*** LLM serving workloads consist of diverse model types with vastly different computational requirements, SLOs, and token length distributions. Hardware device configurations are also heterogeneous in terms of computing power, GPU memory capacity, and GPU-CPU memory bandwidth. To efficiently map LLM requests to the appropriate hardware resources, QLM's global scheduler has to consider each device's computing power, memory capacity, and memory bandwidth. The *RWT estimator* estimates this impact of heterogeneity for the global scheduler. The profiling costs for the RWT estimator are minimal, only a single batch run for a given combination of request group and GPU device is needed. Hence, QLM does not require significant training when adding new LLM models or GPU devices into the serving cluster.

**Algorithm 1** Request Group Creation

```
1: $groups \leftarrow kMeansClustering(requests)$
2: for $i \leftarrow 1$ to $length(groups)$ do
3:     if $groups[i].size() > avg\_batch\_size \times \delta$ then
4:         $newGroups \leftarrow groups[i].splitHalf()$
5:         $groups.append(newGroups)$
6:     end if
7: end for
```

## 4 REQUEST GROUPS AND VIRTUAL QUEUES

In this section, we describe the concept of *virtual queues* and the process of classifying LLM requests into *request groups* and assigning request groups to virtual queues.

**Definition 4.1.** *Request Group:* Each request group is a collection of multiple requests that are relatively homogeneous, i.e., sharing similar performance demand or requirement characteristics. We identify that input/output token distributions, model type, and SLO values are sufficient for the RWT estimator (as explained in Section 6).

**Definition 4.2.** *Virtual Queues:* Each virtual queue is a sequence of request groups that denotes the relative order in which requests will be served. There is a one-to-one mapping between an LLM serving instance and a virtual queue.

By creating the abstraction of request groups and virtual queues, the ordering of request groups in a virtual queue allows QLM to configure actions for multiple downstream LSOs to attain latency SLOs for the LLM-serving requests in a scalable manner (described in Section 5).

**Request Group Creation.** Request groups are created in two steps: (i) clustering similar requests based on Def. 4.1, and (ii) splitting large request groups. Algorithm 1 describes the request group creation process. The parameters identified for the request grouping include model types, input/output token distribution, and SLO values. Grouped requests based on such parameters exhibit predictable request completion time distribution (compared to that of each individual request) as explained in Section 6. Additionally, we also limit the size of each request group to a small multiple ($\delta$) of batch size. We refer the reader to Section 8.3 for the trade-off analysis between overhead and decision-making granularity: (1) Larger request groups would decrease the number of request groups and thus the overhead of the global scheduler; (2) However, restricting the size of request groups is beneficial as it allows for more fine-grained decisions. Since requests within a request group are relatively homogeneous, QLM treats the ordering of the requests within a group using a first-come-first-serve (FCFS) policy. Request groups are dequeued from the virtual queue when all requests complete execution.

**Handling New Incoming Requests.** As new requests join the global queue, they are classified into the existing request groups, and the RWT estimator calculation is triggered to find out whether any SLOs are being violated. Upon any SLO violation, the global scheduler is called to reorder the request groups in the virtual queues to maximize SLO attainment given the current states (estimations).

**Fault Tolerance in Queue Management.** QLM only stores a single replica of the requests and their metadata in the global queue, which avoids the need to maintain consistency between multiple queues. The global queue is implemented using a distributed message queue broker such as RabbitMQ [38] that provides the necessary replication, fault tolerance, and persistence mechanisms. The data structure that implements virtual queues records orderings of subsets of requests in the global queue. These virtual queues are implemented as lightweight data structures that maintain pointers or references to the actual requests stored in the global queue. By using virtual queues, QLM can achieve the following: (1) *Fault Isolation*: If an LLM serving instance fails, only the corresponding virtual queue is affected, and the remaining virtual queues can continue processing requests without interruption. Request groups from the lost virtual queue are assigned to other virtual queues using the global scheduler. (2) *Consistency*: Since the actual requests are stored in the global queue, virtual queues can be reconstructed or reassigned without compromising the consistency of the request data.

## 5 LLM SERVING OPERATIONS

The LLM serving instances serve requests from the corresponding virtual queue and execute backend LSO actions when necessary. The LSOs by themselves are merely action actuators, and the intelligence required to configure when and which action to set comes from the virtual queue ordering set by the global scheduler (as described in Section 7).

Fig. 7 shows the four basic LSOs that QLM currently supports. A QLM agent resident on each LLM serving instance monitors the virtual queue ordering and converts it into LSO actions. When the virtual queue state changes or new requests are added, QLM agents initiate request pulling and load balancing. Similarly, when the head request group changes, the QLM agent initiates request eviction. Model swapping is initiated by the QLM agent for models at the head of the virtual queue. Each of these LSOs modifies the internal state of the LLM serving instance, which includes the running batch of requests, KV cache store, and model weights. Below, we describe each of these LSOs in detail and their action setup based on the virtual queue ordering.

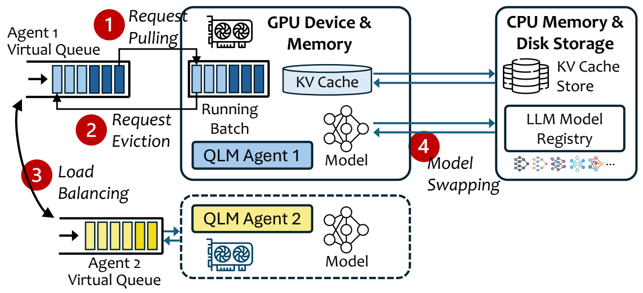

**Figure 7: Basic LLM serving operations (LSOs) for an LLM-serving instance that a QLM agent manages.**

**Request Pulling** (1 in Fig. 7). Request pulling refers to the operation that dequeues the requests in the virtual queue using a pull-based model and adds it to the running batch, i.e., whenever the total tokens of the running batch are below the GPU capacity, a pull signal is issued to retrieve a request from the global queue. The exact pulled request is determined by the request group at the head of the virtual queue. Within the head request group, requests are ordered in an FCFS manner, therefore the first request to join the head request group would be the first to be dequeued from the virtual queue and added to the GPU's running batch. Note that request pulling is insufficient to immediately serve a request with a low SLO value (i.e., stricter SLO) because a pull operation to the virtual queue can only happen if spare token capacity exists on the GPU device (i.e., without head-of-line blocking).

**Request Eviction** (2 in Fig. 7). As request pulling by itself may not be sufficient to enable the immediate serving of requests with low SLO values due to head-of-the-line blocking, QLM also supports request eviction. Request eviction is invoked when the RWT estimator detects an SLO violation, and the global scheduler replaces an existing request group by placing a request group at the head of the virtual queue. In request eviction, requests of the head request group are pulled into the running batch based on available capacity, and previously running requests are evicted (back) into the global queue. To prevent re-computation of KV cache of the lost request upon eviction, we migrate it to CPU memory instead. However, the GPU-to-CPU memory bandwidth is typically at least 10× less than the GPU memory bandwidth, and if the evicted request has a large KV cache, it leads to significant transfer overhead and consequent performance degradation. QLM hides this performance degradation using the asynchronous GPU memory copy available in most GPU programming libraries.

**Load Balancing** (3 in Fig. 7). As each LLM serving instance is associated with a separate virtual queue, the global scheduler's assignment of request groups to a virtual queue inherently performs load balancing. Each instance would only pull from its associated virtual queue, thus ensuring the distribution of requests across all the serving instances. Note that QLM does not implement preemptive load balancing, i.e., once request groups start executing on an LLM serving instance, they cannot be migrated to another instance.

**Model Swapping** (4 in Fig. 7). Each LLM serving instance can serve multiple models by switching the underlying model weights and flushing out the KV cache. QLM assumes a two-tier hierarchy of memory and disk storage. Therefore, any model that needs to be served from the LLM model registry (located in the storage) has to undergo two distinct swaps: (1) *Storage-CPU swapping:* The model is first swapped from the LLM model registry to CPU memory, and (2) *CPU-GPU swapping:* The model located in the CPU memory is swapped into GPU memory for inferencing. QLM is able to decide the location (i.e. GPU mem, CPU mem or storage) of each model by checking the virtual queue order. The model for the head request of the virtual queue is currently *active* and should be placed in the GPU memory. Models present later in the virtual queue are *warm* and placed in the CPU memory until all the CPU memory is exhausted. The remaining models (*cold* models) are not swapped out from the LLM model registry (located in the storage).

**LSO Implementation.** We implement the abovementioned LSOs on top of vLLM, a state-of-the-art LLM serving system. QLM agents are responsible for triggering each LSO action. Request pulling and load balancing are implemented by async pull calls to the virtual queues when there is spare token capacity on the LLM serving instance. Request eviction and GPU-CPU state swapping require instrumentation to the vLLM scheduler. In each iteration, the vLLM scheduler attempts to generate a new token for all the running requests and preempts any request that exceeds the total GPU capacity. At the end of an iteration, QLM agent checks if either request eviction or state swapping is required and performs the operation. For both operations, current requests are removed from the running batch to make space for the incoming requests at the head of the virtual queue. For request eviction, the scheduler uses an asynchronous GPU transfer of the KV cache. Model swapping into the GPU is implemented by changing the underlying model of the vLLM instance and flushing out the KV cache.

## 6 REQUEST WAITING TIME (RWT) ESTIMATOR

QLM leverages waiting time estimates from the RWT estimator for better utilization and decrease resource costs by making closer to optimal request queue reordering decisions. The RWT estimator uses a statistical approach to generate estimates of waiting and completion times for requests. To do so, the estimator first generates an upper bound on request completion time i.e. the actual request completion time would be lesser than the estimate. These completion times are then aggregated to generate completion time of the entire request group. Overall, the estimator is conservative when queue size is small (i.e. the estimate is higher than the actual waiting times) and the accuracy increases with increase in queue size.

**Table 1: Glossary of symbols used in the RWT estimator.**

| Symbol | Description |
|---|---|
| $C_q$ | Completion time for a request $q$ |
| $W_q$ | Request waiting time for a request $q$ |
| $P$ | Prefill time for a request |
| $D_q$ | Total decode time for a request $q$ |
| $O_q$ | Number of output tokens for a request $q$ |
| $\Theta$ | Token generation throughput |
| $\epsilon$ | Inefficiency factor due to continuous batching |
| $d$ | Decode time per output token |

The estimation process is explained in further detail below with variable definitions listed in Table 1.

As shown in Equation 1, the total request completion time equals the sum of the waiting time ($W_q$), prefill time ($P$), and total decode time across all the output tokens ($D_q$) for a request $q$.

$$C_q = W_q + P + D_q \tag{1}$$

**Estimating Prefill Time.** The prefill time $P$ is typically constant per model type when the input tokens are small as it is a highly parallel GPU-accelerated operation whose time increases minimally as the number of input tokens increases [2]. Experiments show that the latency increase from additional input tokens is 100× less compared to the latency increase from each additional output token [2]. Therefore, the major distribution terms that still remain to be estimated are the waiting time ($W_q$) and decode time ($D_q$).

**Estimating Waiting Time.** We consider the token generation throughput ($\Theta$) to be constant throughout the token generation process due to statistical averaging effects described in Appendix A.1. Therefore, the total waiting time for a single request can be represented by Equation 2 by dividing the number of tokens ahead ($\sum_{i=1}^{q-1} O_i$) in the queue by the token generation throughput ($\Theta$) where $i$ denotes each of the $q-1$ requests in the queue ahead of the request we model.

[2]Abnormally long context prompts beyond 10 standard deviations of production prompt distribution are not in the scope of QLM and we leave it for future research work.

$$W_q = \sum_{i=1}^{q-1} \frac{O_i}{\Theta} \quad (2)$$

Note that we do not know the number of output tokens ahead of time (that requires the knowledge of the output sequence for all requests in the waiting queue), so we model them as a distribution with the mean $\mu_o$ and standard deviation $\sigma_o$ fitted from the request input-output history dataset for the request group that the request $q$ belongs to. As $q$ becomes larger, the Central Limit Theorem (CLT) applies and the assumption of Normal distribution is accurate for any underlying request output token distribution. We further explain this in Appendix A.1.

$$\sum_{i=1}^{q-1} O_i \sim N((q-1)\mu_o, (q-1)\sigma_o^2) \quad (3)$$

**Estimating Decode Time.** We compute the total decode time using Equation 4.

$$D_q = O_q \times \epsilon \times d \quad (4)$$

As we do not know the exact number of output tokens ($O_q$) in Equation 4 and the Normal distribution assumption from CLT does not apply for a single request, we approximate it using the maximum possible number of output tokens for the model. As the request queue grows, the waiting time $W_q$ dominates the completion time $C_q$ and the error introduced by the above heuristic reduces as shown in Figure 18. However, for short queues, we maintain the conservative estimate of decode time as it dominates the completion time.

If GPU memory was not a constraint, the decode steps would not be interrupted, and the total decode time would simply be the product of the number of output tokens ($O_q$) and decode time per output token ($d$). However, LLM serving systems cannot ensure this ideal behavior due to continuous batching. As requests are added continuously to the GPU's running batch, some requests inevitably exceed the total GPU memory capacity limit and have to be temporarily preempted. This leads to inefficiency in the generation process that we capture with the inefficiency factor $\epsilon$, i.e., a constant multiplied by the decode time per token that captures the inefficiency associated with the generation process.

Finally, to estimate the completion time of the entire request group (Equation 5), we need to take the max of all the completion times of individual requests.

$$C = \max_q C_q \quad (5)$$

**Table 2: Glossary of symbols used in the linear programming solver.**

| Symbol | Description |
|---|---|
| $g \in \mathbb{G}$ | The $g$-th virtual queue (VQ) in all virtual queues $\mathbb{G}$ |
| $i \in \mathbb{I}$ | The $i$-th request group (RG) in all request groups $\mathbb{I}$ |
| $j$ | Virtual queue position in $[0, L-1]$ with queue length $L$ |
| $x_{g,i,j}$ | Binary decision variable for assignment of RG $i$ to VQ $g$ |
| $wt_{g,j}$ | Request group waiting time |
| $m_{g,j}$ | The model assignment on the $j$-th position of VQ $g$ |
| $t_{g,j}$ | Binary variable for switching the model to serve on VQ $g$ |
| $S$ | Swap time associated with loading a new model into GPU memory |
| $slo_{g,j}$ | SLO preservation rate serving the $j$-th model on VQ $g$ |
| $p_{g,j}$ | Penalty for SLO violation serving the $j$-th model on VQ $g$ |

**Offline Profiling.** There are two independent profiling steps required for the RWT estimator: (a) *Workload Profiling:* samples multiple requests from the workload to generate a distribution for input and output tokens, and (b) *Hardware Profiling:* requires running the model with a single batch of requests on the specific GPU. Fixed variables associated with the model and hardware setup, such as the prefill time ($P$), inefficiency factor ($\epsilon$) and decode time per iteration ($d$) are obtained by directly logging these metrics from the LLM serving instance. In our implementation, we add these logging metrics directly into vLLM code.

## 7 GLOBAL SCHEDULER

The global scheduler is invoked by the RWT estimator when an SLO violation is likely to occur. Upon invocation, the global scheduler runs a linear programming model to reorder the virtual queues that decide underlying LSO actions to maximize SLO attainment. The global scheduler uses a linear program solver because it: (a) allows handling non-determinism by representing request group completion times as distributions, and (b) offers a systematic way to model various constraints associated with SLOs, model swapping times, and hardware heterogeneity. In this section, we present the linear programming model with its defined variables listed in Table 2.

**Overall Modeling Approach.** The goal of the linear programming solver is to find an assignment of request groups to the virtual queues so that all SLOs are met. To model SLO attainment, we define a *penalty* term for each request group, which is the difference between waiting time and SLO value. If SLOs are met, all penalty terms would be smaller than 0. Given the SLOs as the inputs to the linear programming model, we obtain the request group waiting time estimation from the RWT estimator to estimate the defined penalty terms. The worst-case waiting time for a request group is the sum of the waiting time for request groups ahead in the virtual queue from the same model (from Equation 2), the completion time for the request group for different models (from Equation 5), and swap times associated with transferring model weights into GPU memory [3]. Note that effects associated with hardware and model heterogeneity (such as token throughput and eviction vs. swap) that impact request group completion time are captured by the RWT estimator profiling.

**Definitions of Constraints.** Now, we describe each of the constraints in further detail. We assume that each virtual queue can have a maximum length, and every request group is assigned to one of the positions in the virtual queue. Equation 6 models request group assignment to a position in the virtual queue.

$$\sum_g \sum_j x_{g,i,j} = 1 \forall i \qquad \sum_i x_{g,i,j} = 1 \forall g, j \tag{6}$$

Each request group has a one-to-one mapping with a position in a virtual queue. If there are empty positions, we assign them "empty" request groups to match request groups and virtual queue capacity.

Each position in the virtual queue would have a corresponding model and SLO based on Equation 6. This assignment is captured with Equation 7 and Equation 8.

$$m_{g,j} = \sum_i \text{models}_i \times x_{g,i,j} \forall g, j \tag{7}$$

$$slo_{g,j} = \sum_i \text{slos}_i \times x_{g,i,j} \forall g, j \tag{8}$$

The transition between two different models is captured in Equation 9. While inequalities cannot be directly modeled as constraints, we apply the standard big-M method to reduce the inequality into linear constraints [7].

$$t_{g,j} = (m_{g,j-1} \neq m_{g,j}) \forall g, j \tag{9}$$

The cumulative waiting times of all positions in the virtual queue would be the sum of waiting time, completion times, and swap times as represented in Equation 10. $C$ is the completion time of a request group from Equation 5.

$$\text{wt}_{g,j} = \sum_i \sum_k^{j-1} W_{g,i} \times x_{g,i,k} + \sum_k^{j-1} t_{g,k} \times S + \sum_i \sum_k^{j-1} C_{g,i} \times t_{g,k} \times x_{g,i,j} \forall g, j \tag{10}$$

The penalty would simply be the difference between the waiting time and the SLO value, as shown in Equation 11.

$$p_{g,j} = wt_{g,j} - slo_{g,j} \forall g, j \tag{11}$$

The final constraint is that all penalty values should be less than 0 i.e., all SLOs are satisfied.

$$p_{g,j} \leq 0 \forall g, j \tag{12}$$

**Optimization Goal.** The linear programming model aims to minimize the total penalty for SLO violations.

$$\min(\sum_g \sum_j p_{g,j}) \tag{13}$$

## 8 EVALUATION

Our experiments address the following research questions:

(a) QLM performance with respect to SLO attainment and request throughput in single-model serving (Section 8.1),
(b) QLM performance with respect to SLO attainment and request throughput in multi-model serving (Section 8.2),
(c) Contribution of each LSO to QLM performance,
(d) Accuracy of the RWT estimator in request waiting time estimation,
(e) Robustness analysis of QLM to hardware heterogeneity, token distributions, burstiness, and request group size regarding LLM-serving performance (Section 8.3), and
(f) Overhead of using QLM with increasing queue sizes.

**Experiment Setup.** We evaluate QLM on multiple varying-sized open-source LLMs: Mistral-7B [17], Vicuna-13B [5], and Llama-70B [47]. We evaluate on a test bed consisting of GPUs of two types: 30 NVIDIA A10 (24 GB memory) and 50 NVIDIA A100 (80 GB memory). The setup represents both model and device heterogeneity. To evaluate the benefit of QLM, we consider the following three baseline mechanisms: (1) *EDF* (Earliest Deadline First): Requests are sorted by their SLO values such that requests with the smallest SLO values are at the front of the virtual queue, (2) *vLLM* [20]: Requests use the default first-come-first-serve (FCFS) scheduler in vLLM, and (3) *SHEPHERD* [54]: Requests are served with dynamic batching and an ILP formulation for ordering and placement. Note that SHEPHERD cannot be easily extended to work with continuous batching as the LP formulation assumes fixed batches with deterministic execution times.

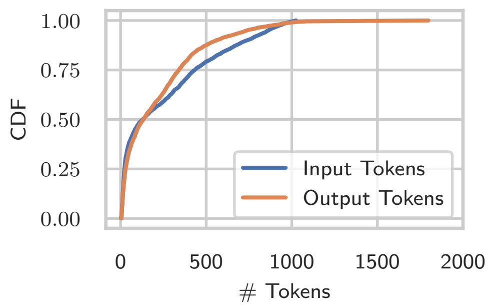

**Figure 8: Distribution of input and output tokens in the shareGPT dataset.**

**Workloads.** We create our experimental workloads from the requirements of a production cloud service provider except for request arrival rates due to confidentiality reasons.

[3]We measure swap times from model load time profiling.

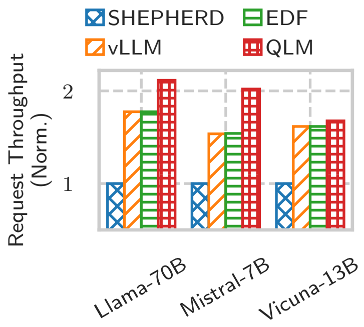

**Figure 9: Single model request serving throughput at 0.5K requests/s interactive arrival rate. Increased throughput corresponds to 1.1-2.3× GPU requirement reduction.**

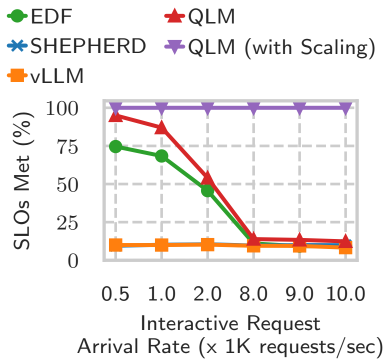

**Figure 10: Single model SLO satisfaction for varying interactive request arrival rates for Vicuna 13B.**

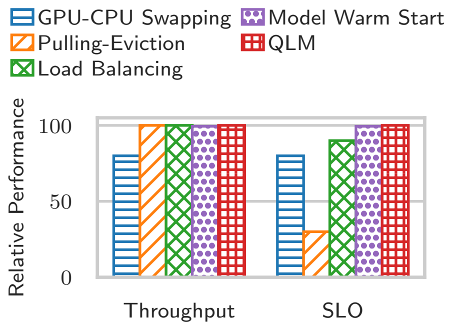

**Figure 11: Single model LSO ablation study at 0.5K requests/s interactive arrival rate for Vicuna 13B.**

Request arrivals are modeled with a Poisson distribution and queues are created by varying the arrival rates. Each workload trace uses 3,500 requests from the ShareGPT [46] dataset with input/output token distribution as shown in Figure 8. We classify all requests into three categories and define their SLO values accordingly: (1) *Interactive:* 20s, (2) *Batch-1:* 1 min, and (3) *Batch-2:* 1 hour. Note that these SLOs are defined with respect to the 99th percentile value of the time to first token (TTFT). We test the experimental workload in the following three scenarios: **[$W_A$]** ***Single-Model Interactive and Batch Workload*** which consists of Batch-1, Batch-2, and Interactive requests for a single model, and no model swapping is required. **[$W_B$]** ***Multi-Model Batch Workload*** which consists of Batch-1 and Batch-2 requests. Batch-1 requests use two models: fine-tuned versions of Mistral 7B and Llama 70B. Batch-2 requests use three models: fine tuned versions of Vicuna 13B and Llama 70B. **[$W_C$]** ***Single-Model MegaPrompt Workload*** This request workload consists of several "mega prompts" in addition to the workload from $W_B$. To generate the mega prompt workload, we randomly select requests with total input and output tokens in the 3K – 4K range. These mega prompts have a large number of input and output tokens that occupy a large percentage of GPU memory and cause further HOL blocking.

We do not present results for the multi-model interactive workload because QLM assigns a separate GPU for each model, effectively reducing the workload to a single model workload ($W_A$). QLM' global scheduler decides not to swap the models as the model swapping time exceeds the interactive request SLO (20s).

## 8.1 Single-Model Evaluation

We run workload $W_A$ on 50 A100 GPUs to evaluate the *single-model* LLM serving performance regarding the request throughput, SLO attainment, and LSO contribution ablation study (similar to the multi-model evaluation in Section 8.2).

**Request Throughput and SLO Attainment.** Figure 10 shows the percentage of SLOs that are satisfied by QLM and the baseline systems. We find that when the request arrival rate significantly exceeds the serving capacity, none of the systems can satisfy the SLOs. This is because the minimum serving time is much longer than the specified SLO. As the arrival rate of interactive requests decreases, QLM performs the best in satisfying the maximum number of SLOs. Specifically, it performs better than the baseline mechanisms because: (a) Compared to vLLM, QLM is able to move interactive service requests ahead in the queue, (b) Compared to EDF, QLM enables appropriate eviction of batch requests from the running queue, and (c) Compared to SHEPHERD, QLM uses continuous batching as opposed to static batch size and models the auto-regressive LLM nature with the RWT estimator to increase request throughput.

We find that the advantages of QLM with respect to smart selection among various LSOs, continuous batching, and appropriate request prioritization help with improving request throughput. Figure 9 shows the request throughput for QLM and the individual baseline mechanisms at arrival rate of 0.5K requests/s where QLM is able to achieve the maximum SLO satisfaction. QLM achieves higher throughput, i.e., 20% higher compared to vLLM and EDF, and 50% higher than SHEPHERD.

**Contribution of Each LSO.** Figure 11 shows the impact of removing each LSO considered by the backend LLM serving

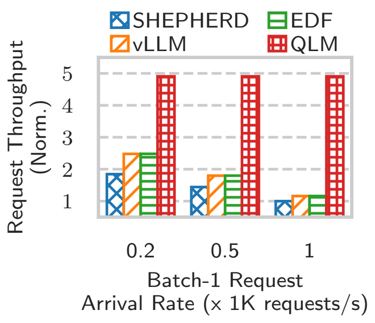

**Figure 12: Multi-model request serving throughput for varying Batch-1 request arrival rates. Increased throughput corresponds to 2-5× GPU requirement reduction.**

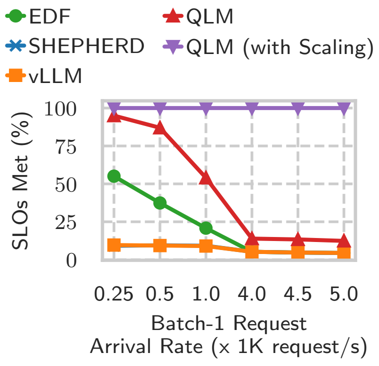

**Figure 13: Multi-model SLO satisfaction for varying Batch-1 request arrival rates.**

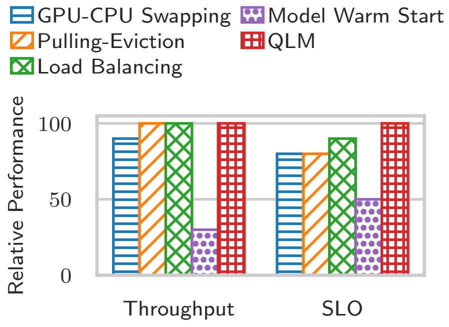

**Figure 14: Multi-model LSO ablation study for 0.25K requests/sec Batch-1 arrival rate.**

instance in QLM. Request pulling contribute significantly to latency reduction for interactive services and consequently increase the number of SLOs met. Request eviction increases request throughput by swapping the KV cache into CPU memory. Finally, model swapping has no impact on this workload as a single model is being served.

## 8.2 Multi-Model Evaluation

We run workload $W_B$ to evaluate the *multi-model* performance on 50 A100 GPUs with respect to request throughput and SLO satisfaction. Additionally, we also provide an ablation study to understand the contribution of each LSO to the overall QLM performance.

**Request Throughput and SLO Attainment.** Figure 12 shows the request throughput (i.e., requests served per second) for $W_B$ comparing QLM with the baseline mechanisms for varying Batch-1 arrival rates. QLM provides up to 3–4× higher throughput due to the following factors: (1) The use of request groups minimizes repeated swapping required as the model would only be swapped in once per request group instead of per individual request, and (2) The global scheduler couples every Batch-1 model with another Batch-2 model to minimize swaps while maintaining an equal distribution of queue sizes.

The improvement in request throughput directly maximizes the percentage of SLO satisfied for all requests. Figure 13 shows the percentage of SLO satisfied for the interactive services against against their arrival rate. When the Batch-1 requests arrive at less than 0.5K requests/s, QLM satisfies more than 90% of all SLO values. As the arrival rate of Batch-1 service requests increases, no combination of requests would be able to meet all SLOs, and the global scheduler would fail to return a solution. In such a scenario, a scale-up action is required to add more GPU devices. We perform this scale-up action to enable 100% SLO attainment if the current GPU capacity is insufficient. The baselines perform worse compared to QLM because none of them consider the impact of model swapping. Other limitations of the baselines are discussed in Section 8.1.

**Contribution of Each LSO.** Each of the four LSOs used by QLM, including request pulling, request eviction, model swapping, and load balancing, contributes to either the latency and/or the throughput of the serving system. Figure 14 shows the impact of removing each LSO on QLM performance for $W_B$. The model warm start LSO contributes the most to QLM performance for both SLOs and throughput, as multiple models need to be multiplexed on the same LLM serving instance. Additionally, the other LSOs contribute primarily to the latency SLO attainment.

## 8.3 QLM Robustness Analysis

**Hardware Heterogeneity.** We run $W_A$ on a mix of A10 and A100 GPUs to evaluate the robustness of QLM performance in heterogeneous hardware setup. Figure 15 shows request throughput when the cluster has varying ratios of A10 to A100 GPUs. The A10 is a lower-end GPU with ~3× lower GPU memory and thus is only capable of serving a much lower request throughput compared to the A100 GPU. QLM takes into account this difference between request throughput across GPUs via the RWT estimator with offline profiling, and the global scheduler proportionally assigns a lower number of requests to the A10 GPU compared to the A100. On the other hand, if we use a round-robin policy for request

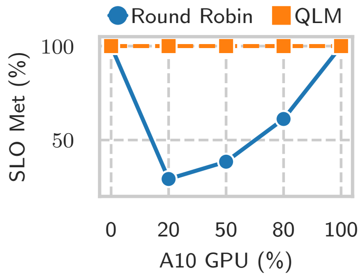

**Figure 15: Impact of hardware heterogeneity.**

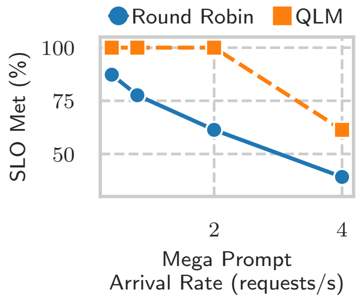

**Figure 16: Impact of mega prompt arrivals.**

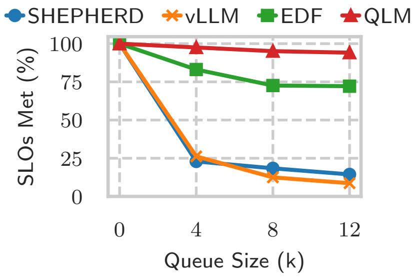

**Figure 17: Impact of increasing queue size on SLO satisfaction.**

assignment to the LLM serving instances (while using default QLM policy per instance), the load would be distributed equally, leading to higher queue drain times for the A10 GPU. Additionally, we also observe that the benefit of QLM is more compared to a random policy when the heterogeneity of the cluster is higher. When the A10 GPUs constitute 20−50% of the cluster (more heterogeneous), the improvement of QLM over random policy is 2−5× higher compared to a 100% A10 or 100% A100 composition (more homogeneous). Note that for experimental purposes, we increase the total number of A100s proportionately to demonstrate the impact of GPU imbalance.

**Mega Prompt Workload.** The RWT estimator of QLM takes into account input and output token distribution when estimating the request waiting time. Consequently, when there are distinct token distributions, such as in workload setup $W_C$, QLM is able to load balance them intelligently across LLM serving instances to minimize the queue drain time. For example, in workload $W_C$, the "mega prompts" use a large number of tokens, and their KV cache occupies the entire GPU memory, causing head-of-the-line blocking for the regular requests in the queue. The optimal policy, as identified by QLM, in such a scenario would be to allocate all the regular requests to another LLM serving instance. Note that request eviction is not an option if all SLOs are tight. Figure 16 shows the benefit of QLM for workload $W_C$. The relative benefit of QLM is highest for a few mega prompts because the regular requests can be moved to another GPU. As the percentage of mega prompts increases, there is no option but to assign them to different LLM serving instances, causing inevitable HOL blocking, and the benefit of QLM reduces. In such a case, we would need to perform a scale-up action and add more GPU devices to the cluster to continue maintaining SLOs.

**Varying Queue Size and Burstiness.** The benefit of QLM is largely present when the queue size is large, and intelligent decision-making is required for setting LSO actions. Thus, to show the benefit of QLM under varying queue sizes, we vary the arrival rates of requests in $W_B$ to create a large queue and compare it against the baseline systems as shown in Figure 17. When the queue size is 0, QLM offers no benefit in maintaining SLOs as compared to the baseline approaches because the system is underutilized and does not require any smart decision-making. However, as the queue size increases, the percentage of SLOs met by the baseline systems keeps dropping due to reasons described in Section 8.1, while QLM is able to maintain a high SLO satisfaction percentage.

**RWT Estimator Accuracy.** The RWT estimator calculates the request waiting time based on initial profiling of the model and hardware setup. This initial profiling time is negligible as only a single batch of requests needs to be run on the GPU.

Figure 18 shows the coefficient of determination ($R^2$ values) when estimating waiting times for increasing queue sizes across the different models. Overall, we confirm our observation that as the number of request groups in the queue increase the waiting time estimation becomes more accurate. Specifically, we find with four request groups the RWT estimator reaches an accuracy of 0.99.

While the RWT estimator is highly accurate in estimating request waiting time for longer request queues, it is not perfect. When request queues are small, statistical averaging effects of continuous batching do not hold and the waiting time estimation tends calculates more conservative (i.e. higher) estimates leading to lower estimation accuracy.

**Impact of Request Group Size.** QLM sets the request group size as a multiple ($\delta$) of the average batch size. The exact $\delta$ value depends on the acceptable trade-off between the overhead of running the global scheduler and the granularity of decision-making. As $\delta$ becomes smaller, QLM achieves a finer granularity of decision-making, leading to improved performance. However, the overhead leads to delayed decision-making. Figure 19 demonstrates this tradeoff between performance degradation (caused by changing granularity in

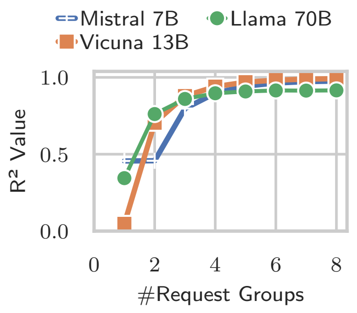

Figure 18: Accuracy of RWT estimator.

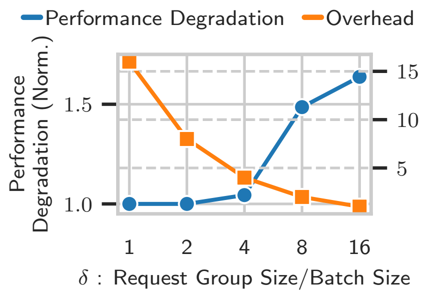

Figure 19: Impact of request group size on QLM performance.

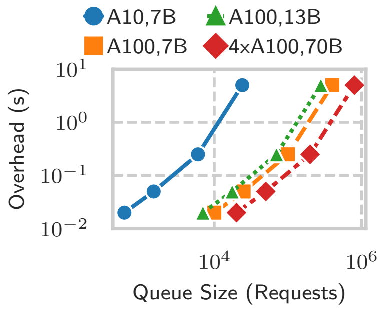

Figure 20: QLM Overhead.

decision making) and overhead of the global scheduler when varying $\delta$. At $\delta = 16$, the overhead is smallest, but decision-making granularity is coarse, leading to sub-optimal decisions (such as imbalance between virtual queue sizes of LLM serving instances). In contrast, at $\delta = 1$, the performance degradation is minimal, but overhead is much higher. We choose $\delta = 4$, as it results in nearly zero performance degradation, compared to $\delta = 1$, while maintaining a low overhead.

**Scalability and Overhead.** The global scheduler is invoked only when QLM detects an SLO violation using the RWT estimator. While the global scheduler is performing the virtual queue reordering, QLM continues serving and requests with tight SLOs (low latency requirements) placed at the front of the virtual queues are not interrupted. This ensures the global scheduler is off the critical serving path (i.e. the overheads can be hidden) and ensures SLO compliance. While the global scheduler is the primary overhead, other system overheads include the CPU memory required for the request eviction and model swapping LSOs. Specifically, we require an additional 80 GB CPU memory for Vicuna 13B and Mistral 7B, and 320 GB CPU memory for Llama 70B.

In Figure 20, we show the time required to solve for the linear program in the global scheduler with varying queue sizes in terms of the number of requests. As the basic unit of the solver is a single request group, the model and GPU configurations with a larger request group size would be able to handle a much larger queue size for the same overhead. Consequently, configurations with a large request group size, such as an A100 with a 7B model, can handle a maximum queue size of 400K requests at a 5s overhead per request group (i.e., 5 ms per request, a production requirement).

## 9 DISCUSSION AND FUTURE WORK

**What are the failure scenarios for the RWT Estimator?** The RWT Estimator has the following drawbacks: (a) If the number of request groups in the queue are small, QLM overestimates the request waiting time. Such a conservative estimate is important as it would still lead to SLO preservation by the global scheduler, albeit the system could be underutilized compared to optimal. (b) When the number of output tokens is out-of-distribution (for example, more than 5-10 standard deviations above mean), the estimator would underestimate the request group completion time leading to potential SLO violations. However, in our experience with shareGPT and production traces such scenarios are rare.

**What happens if QLM is unable to meet SLOs?** QLM's global scheduler may not be able to find an optimal virtual queue ordering if the request demand is high and the number of LLM serving instances (i.e., underlying compute resources) is insufficient. In such cases, we have three choices: (a) scale up the number of LLM serving instances by adding GPU devices, as we demonstrate in Figure 13 and Figure 10, (b) fall back to a heuristic such as Earliest Deadline First (EDF) and continue serving requests, and (c) performance admission control or rate limiting by dropping incoming requests to limit queue size. Option (a) can only be used when there are available resources, whereas Option (b) and Option (c) would lead to SLO violations.

**Can new LSOs be added to QLM?** QLM can be extended to support other LSOs that depend on the queue size and request ordering. For example, GPU partitioning techniques (such as NVIDIA MIG [30]) can be added as an LSO with additional constraints on memory in the linear programming solver described in Section 7. We leave the addition of extra LSOs to our future work.

**How can QLM handle request priorities?** QLM can also be used when strict priorities are assigned to requests instead of SLO values. In the strict priority model, $request_1$ would execute before $request_2$, if $priority(request_1)$ is less than $priority(request_2)$. With strict priority, the relative ordering of requests across priorities is pre-decided, however

| **Category** | **Multi Instance** | **SLO-aware** | **Autoregressive Optimizations** | **Hardware Heterogeneity** |
|---|---|---|---|---|
| LLM Scheduling Optimizations [16, 20, 51] | ✗ | ✗ | ✓ | ✓ |
| LLM Serving Backend Optimizations [1, 10, 11, 22, 24, 26–28, 34, 42, 43, 55–57] | ✗ | ✗ | ✓ | ✓ |
| General ML Model-Serving Systems [8, 9, 13, 53, 54] | ✓ | ✓ | ✗ | ✓ |
| LLM Orchestration Systems [45] | ✓ | ✗ | ✓ | ✗ |
| **QLM** | ✓ | ✓ | ✓ | ✓ |

✓: Supported, ✗: Not Supported

**Table 3: Comparison of QLM with Related Work**

per-priority assignment still needs to be optimized to maximize SLO satisfaction. Consequently, the concepts of virtual queues, request groups, and the RWT estimator continue to remain useful.

**Can SLOs be defined on end-to-end latency?** QLM addresses the problem of optimizing time to first token (TTFT) SLO attainment. However, it can also be made to work with the modified objective of end to end request completion time SLOs (i.e. including decoding iterations). In this case, the RWT estimator and global scheduler need to be modified to consider a distribution of number of output tokens per request (as the exact number are unknown aprioiri).

**Can SLOs be defined on inter-token latency?** QLM primarily addresses the problem of TTFT and does not guarantee the inter-token latency (ITL). However, related work such as Andes [25] attempts to guarantee token generation speed (that can be converted into inter-token latency) by modifying the local vLLM scheduler. QLM can work together with such systems to ensure ITL SLOs along with TTFT SLOs.

## 10 RELATED WORK

**LLM Scheduling and Orchestration.** Existing state-of-the-art LLM serving systems [16, 20, 51] adopts continuous batching and a first-come-first-serve (FCFS) scheduling policy that suffers from head-of-line (HOL) blocking, which we address in QLM. FastServe [49] proposes preemptive scheduling with a Multi-Level Feedback Queue. Andes [25] defines Quality-of-Experience (QoE) for LLM serving as token delivery speed, and proposes a preemptive scheduler that maximizes QoE. Llumnix [45] is an orchestration system across multiple LLM serving instances. QLM is the first queue management framework that optimizes SLO attainment while improving LLM-serving throughput and device utilization by systematically orchestrating backend LSOs.

**LLM Serving Backend Optimization.** Various LLM serving backend optimization techniques have been proposed to improve token generation throughput and memory cost while adapting to fine-tuning paradigms such StreamingLLM, Speculative Decoding, ChunkedAttention, FlashAttention and more [1, 10, 11, 22, 24, 26–28, 34, 42, 43, 55–57]. These backend LLM-serving optimizations are complementary to QLM as the LLM serving instance (see Def. 2.3), and their impact on token generation throughput can be captured with profiling for the RCT Estimator (see Section 6).

**General ML Model-Serving Systems.** Traditional model-serving systems provide functionalities such as scheduling, placement, batching, and autoscaling. Clipper [9], TensorFlow-Serving [32], MArk [53], InferLine [8], SHEPHERD [54], and Clockwork [13] are some earlier work on serving traditional ML models like ResNet that are relatively small. INFaaS [39] and Cocktail [14] propose a model-less serving framework to automate the model selection and autoscaling to meet SLOs. However, they fail to consider the autoregressive property of LLMs. On the other hand, advanced autoscaling techniques are complementary to QLM.

## 11 CONCLUSION

We presented QLM, a novel queue management framework that orchestrates backend LSOs for SLO-oriented LLM serving. Evaluation using real-world LLM serving datasets on heterogeneous model types and GPU devices demonstrate that QLM improves end-to-end latency SLO attainment by 40–90% while improving serving throughput and device utilization by 20-400%.

## 12 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for providing their valuable feedback. We also thank Nick Hill, Lionel Villard, and Mudhakar Srivatsa for providing us with production requirements and insight, and Rohan Arora for infrastructure support. This work is supported by the National Science Foundation (NSF) under grant No. CCF 20-29049 and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

## REFERENCES

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. APIServe: Efficient API Support for Large-Language Model Inferencing. arXiv:arXiv preprint arXiv:2402.01869
[2] Anyscale. 2024. Reproducible Performance Metrics for LLM inference. https://www.anyscale.com/blog/reproducible-performance-metrics-for-llm-inference. Accessed: 2024/04/10.
[3] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the Opportunities and Risks of Foundation Models. arXiv:arXiv preprint arXiv:2108.07258
[4] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. Parties: QoS-aware resource partitioning for multiple interactive services. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*. ACM, Providence, RI, 107–120.
[5] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90% ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/
[6] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on multi-GPU servers with Spatio-Temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 2022)*. USENIX, Carlsbad, CA, 199–216.
[7] Marco Cococcioni and Lorenzo Fiaschi. 2021. The Big-M method with the numerical infinite M. *Optimization Letters* 15, 7 (2021), 2455–2468.
[8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285
[9] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*. USENIX, Boston,MA, 613–627.
[10] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: An Efficient GPU Serving System for Transformer Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2021)*. ACM, Austin, TX, 389–402.
[11] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2024. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. arXiv:2310.01801 [cs.CL]
[12] Ulrich Gnewuch, Stefan Morana, Marc TP Adam, and Alexander Maedche. 2022. Opposing Effects of Response Time in Human–Chatbot Interaction: the moderating role of prior experience. *Business & Information Systems Engineering* 64, 6 (2022), 773–791.
[13] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. USENIX, Berkeley, CA, 443–462.
[14] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*. USENIX, Renton, WA, 1041–1057.
[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*. IEEE, Las Vegas, NV, 770–778.
[16] HuggingFace. 2024. *Text Generation Inference*. HuggingFace. Retrieved July 1, 2024 from https://github.com/huggingface/text-generation-inference
[17] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B.
[18] Hanlei Jin, Yang Zhang, Dan Meng, Jun Wang, and Jinghua Tan. 2024. A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods.
[19] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. 2016. Morpheus: Towards automated SLO for enterprise clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*. USENIX, Savannah, GA, 117–134.
[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 (SOSP 2023)*. ACM, Koblenz, Germany, 611–626.
[21] Matthew LeMay, Shijian Li, and Tian Guo. 2020. Perseus: Characterizing performance and cost of multi-tenant serving for CNN models. In *2020 IEEE International Conference on Cloud Engineering (IC2E 2020)*. IEEE, IEEE, Boston, MA, 66–72.
[22] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. PMLR, PMLR, Honolulu, HI, 19274–19286.
[23] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2023)*. USENIX, Boston, MA, 663–679.
[24] Pierre Lienhart. 2024. LLM Inference Series: 4. KV caching, a deeper look. https://medium.com/@plienhar/llm-inference-series-4-kv-caching-a-deeper-look-4ba9a77746c8 (Accessed on 04/10/2024).
[25] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. 2024. Andes: Defining and Enhancing Quality-of-Experience in LLM-Based Text Streaming Services.

[26] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. 2023. Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. arXiv:2305.17118 [cs.LG]

[27] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. 2023. DejaVu: Contextual Sparsity for Efficient LLMs at Inference Time. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. PMLR, PMLR, Honolulu, HI, 22137–22176.

[28] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2023. SpotServe: Serving generative large language models on preemptible instances.

[29] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data?

[30] NVIDIA. 2024. Nvidia Multi-instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. Accessed: 2024/04/10.

[31] NVIDIA. 2024. *TensorRT-LLM*. NVIDIA. Retrieved July 1, 2024 from https://github.com/NVIDIA/TensorRT-LLM

[32] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving.

[33] OpenAI. 2024. *OpenAI - Finetuning*. OpenAI. Retrieved July 1, 2024 from https://platform.openai.com/docs/guides/fine-tuning

[34] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. 2023. Splitwise: Efficient generative LLM inference using phase splitting. arXiv:2311.18677 [cs.AR]

[35] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *Proceedings of The 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. USENIX, Virtual, 805–825.

[36] 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. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, Vol. 5. Association for Computing Machinery, San Diego, CA, USA, 1–7.

[37] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2024. Power-aware Deep Learning Model Serving with $\mu$-Serve. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 2024)*. USENIX, Santa Clara, CA, 75–93.

[38] RabbitMQ. 2024. RabbitMQ. https://www.rabbitmq.com/.

[39] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of 2021 USENIX Annual Technical Conference (ATC 2021)*. USENIX, Virtual, 397–411.

[40] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code.

[41] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. 2017. Recent advances in recurrent neural networks.

[42] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2023. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. arXiv:2311.03285 [cs.LG]

[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. PMLR, PMLR, Honolulu, HI, 31094–31116.

[44] Shivchander Sudalairaj, Abhishek Bhandwaldar, Aldo Pareja, Kai Xu, David D Cox, and Akash Srivastava. 2024. Lab: Large-scale alignment for chatbots.

[45] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving.

[46] Vicuna team. 2024. *ShareGPT Dataset*. Vicuna team. Retrieved July 1, 2024 from https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models.

[48] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2024. Towards Efficient and Reliable LLM Serving: A Real-World Workload Study. arXiv:2401.17644 [cs.DC]

[49] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models.

[50] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework.

[51] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022)*. USENIX, Carlsbad, CA, 521–538.

[52] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

[53] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC 2019)*. USENIX, Renton, WA, 1049–1062.

[54] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. Shepherd: Serving DNNs in the Wild. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023)*. USENIX, Boston, MA, 787–808.

[55] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. $H_2O$: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. arXiv:2306.14048 [cs.LG]

[56] 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. arXiv:2401.09670 [cs.DC]

[57] Lei Zhu, Xinjiang Wang, Wayne Zhang, and Rynson W. H. Lau. 2024. RelayAttention for Efficient Large Language Model Serving with Long System Prompts. arXiv:2402.14808 [cs.CL]

| Symbol | Description |
|---|---|
| $W$ | Total waiting time for a request in the queue |
| $O$ | Total number of output tokens in the queue |
| $\Theta$ | Output token generation throughput (tokens/s) |
| $B$ | Batch size |
| $\delta$ | Decoding time per token |
| $\epsilon$ | Preemption factor |
| GPU | Total token memory capacity in GPU |
| $I_i$ | Number of input tokens for the $i$-th request |
| $O_i$ | Number of output tokens for the $i$-th request |
| $\mu_O$ | Mean number of output tokens per request |
| $\sigma_O^2$ | Variance of output tokens per request |
| $\mu_I$ | Mean number of input tokens per request |
| $\sigma_I^2$ | Variance of input tokens per request |
| n | Size of the waiting queue |

**Table 4: Glossary of Symbols Used in the Statistical Derivation**

## A APPENDIX

### A.1 Waiting Time Predictability

Below, we present a statistical derivation for the underlying assumptions from the RWT estimator from Section 6.

The total number of output tokens in the queue would be the sum of output tokens of individual requests. The total waiting time for a request joining such a queue would be the time to process output tokens for requests ahead in the queue, which would be the number of output tokens divided by the token generation throughput.

$$W = \frac{O}{\Theta} \tag{14}$$

The average token generation throughput (i.e. the number of output tokens generated per second) is simply the average batch size divided by time to generate each token.

$$\Theta = \frac{B}{\delta \times \epsilon} \tag{15}$$

Due to continuous batching, the batch size is simply the number of tokens that can be kept in GPU memory.

$$B \approx \frac{\text{GPU}}{I_i + O_i} \tag{16}$$

Simplifying the above equations, we derive the waiting time to be:

$$W = \frac{O \times \delta \times \epsilon \times E[I_i + O_i]}{GPU} \tag{17}$$

Therefore the expected waiting time would be:

$$E[W] = \frac{E[O] \times \delta \times \epsilon \times E[I_i + O_i]}{GPU} \tag{18}$$

As each request is independent of others and all requests originate from the same model, we assume the number of tokens for requests in the batch and queue to be i.i.d. (independent and identically distributed) variables.

Due to the i.i.d. condition, for a large queue, we can apply the Central Limit Theorem and approximate the total number of output tokens with a normal distribution.

$$O = \mathcal{N}(n\mu_O, n\sigma_O^2) \tag{19}$$

$$E[O] = n\mu_O \tag{20}$$

Similarly for a large batch, we can also apply the Central Limit Theorem to approximate the number tokens in a batch with a normal distribution.

$$I_i + O_i = \mathcal{N}(\mu_{I+O}, \sigma_{I+O}^2/B) \tag{21}$$

$$E[I_i + O_i] = \mu_I + \mu_O \tag{22}$$

As remaining terms in the waiting time estimation are constant, we find that the expected waiting time in the queue can be estimated analytically.