One Queue Is All You Need: Resolving Head-of-Line Blocking in Large Language Model Serving

Archit Patke[1] Dhemath Reddy[1] Saurabh Jha[2] Haoran Qiu[1] Christian Pinto[3]
Shengkun Cui[1] Chandra Narayanaswami[2] Zbigniew Kalbarczyk[1] Ravishankar Iyer[1]

[1]*University of Illinois at Urbana-Champaign* [2]*IBM Research* [3]*IBM Research Europe*

Abstract

Large language models (LLMs) have become an increasingly important workload for cloud providers catering to both enterprise and consumer applications. LLM inference requests from these applications have *end-to-end latency SLOs* that must be adhered to in production settings. However, existing LLM serving systems focus on optimization objectives such as request serving throughput or request execution latency rather than the end-to-end latency SLOs. Achieving end-to-end SLOs for latency-sensitive requests is challenging due to *head-of-line (HOL)* blocking in the request queue, which results from bursty arrival rates and insufficient resources.

To address the above challenge, we propose QLM, a multi-model queue management framework for LLM serving. QLM uses stochastic programming to orchestrate the actions of multiple LLM Serving Operations (LSOs) to reduce HOL blocking and maximize SLO attainment. Specifically, QLM uses the following LSOs: model swapping, request eviction, GPU-CPU state swapping, load balancing, and warm model start. 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.

1 Introduction

Motivation. The emergence of large language models (LLMs) such as OpenAI GPT-4 and Google Gemini has enabled a wide range of novel AI applications [3,56,58], including chatbots and coding assistants. Consequently, serving LLMs has become an increasingly critical workload, catering to both enterprise and consumer applications with service-level objectives (SLOs) on end-to-end latency [20,37,53]. However, existing LLM-serving systems [21,25,38,45,48,49,55,57] focus on optimization objectives such as request serving throughput, device utilization, or request execution latency rather than addressing SLOs on the *end-to-end latency*, which encompasses both request *execution time* and *waiting time*. Optimizing end-to-end latency SLOs is critical as it is the prime metric valued by the customers using LLM inferencing services [30,31].

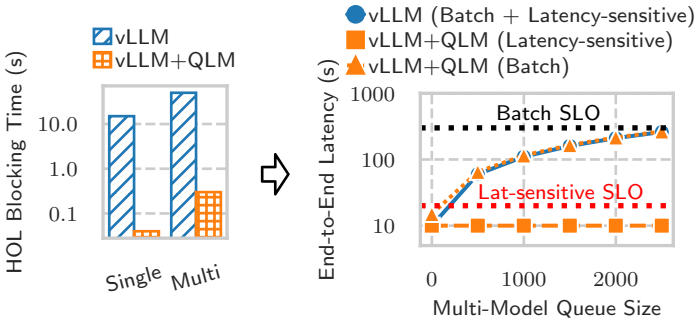

Figure 1: QLM nearly eliminates HOL blocking time in both single and multi-model serving scenarios and enables SLO attainment for batch and latency-sensitive applications.

When maximizing end-to-end latency SLO attainment, minimizing request waiting time is just as important as execution time due to the presence of *request queues*. Request queues fill up when the serving throughput is unable to match the high and bursty arrival rates of the incoming requests [53]. The issue of increasing request queue sizes is further exacerbated by device shortage and the need to serve multiple fine-tuned heterogeneous models with varying sizes [24], resulting in high cold start time [12] and low device utilization [62]. Latency-sensitive requests with strict SLOs may wait longer in the queue because of *head-of-line (HOL) blocking*, leading to potential SLO violations. Figure 1 (left) shows that requests can suffer from HOL blocking in both single and multi-model serving using production LLM-serving traces [36] (labeled as "vLLM", a state-of-the-art LLM serving system).

Our Work. Fundamentally, HOL blocking can be alleviated by reordering the requests in the waiting queue and assigning requests to appropriate devices. In the LLM serving context, this assignment and reordering can be mapped to *LLM serving operations (LSOs)* such as request pulling, request eviction, load balancing, GPU-CPU state swap, and model warm start. To orchestrate these LSOs and maximize SLO attainment, we

introduce QLM, a multi-model queue management system. Figure 1 (right) shows that QLM maximizes SLO attainment for both latency-sensitive and batch jobs in a multi-model serving workload setup. QLM leverages two major ideas in its queue management framework:

(a) *Virtual Queue Abstraction:* Previous work tends to optimize LSOs separately and in an ad-hoc manner [15, 46, 54]. An LLM serving framework that systematically investigates the trade-offs amongst multiple LSOs for SLO attainment is missing to date. To bridge this gap, QLM introduces the concept of *virtual queues* that represent the *dynamic order and assignment of requests* to be served, which provides a necessary abstraction for optimizing LSO actions. Moreover, selecting optimal LSOs per request is an NP-hard decision problem that will not meet the acceptable overhead of executing requests in production (e.g., < 10 ms per request). To reduce the complexity of the decision problem, QLM uses *request groups*, where requests that share the same performance characteristics are grouped together, and decisions are taken per request group instead of per request.

(b) *Plan Generator and Request Completion Time (RCT) Estimator:* When making decisions on request group ordering in the virtual queues, the two key metrics that need to be estimated are the request waiting time and execution time. However, estimating these metrics is challenging because the execution time for a request depends on its output token length, which is unknown apriori before executing the request [21] (i.e., stochastic in nature). QLM models this stochastic nature and estimates the waiting and execution time distributions using a *Request Completion Time (RCT) estimator* (described in Section 6). Finally, QLM uses a *Plan Generator* (described in Section 7) that leverages the completion time distribution of request groups to create an optimal ordering and assignment of request groups onto the virtual queues to maximize SLO attainment. Because the key variables in the optimization, output token length and completion times are stochastic variables, the plan generator is based on a *stochastic programming solver*. In comparison, other ML serving systems such as Clockwork [16] and SHEPHERD [60] use variations of linear programming solvers because the request completion time and waiting time are deterministic for traditional ML models (e.g., ResNet).

In summary, QLM enables the translation of end-to-end per-request latency SLOs into backend LSO actions using a combination of virtual queues, stochastic modeling of queue dynamics (in RCT estimator), and stochastic programming (in plan generator).

Results. We demonstrate QLM on our internal production-grade version of vLLM [21] as the backend LLM-serving system. QLM supports the following five basic LSOs (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) *GPU-CPU Swapping* for the internal LLM state, (4) *Model Warm Start* from CPU memory instead of disk, and (5) *Load Balancing* across multiple LLM model instances.

We evaluate QLM on three popular LLMs of varying sizes (i.e., Mistral-7B [19], Vicuna-13B [5], and Llama-70B [50]) on heterogeneous GPU clusters with NVIDIA A10 and A100 GPUs. We adopt workloads from a real-world LLM dataset: ShareGPT [43] using setups derived from our production requirements. Our experiments demonstrate the following major improvements with QLM:
(1) *SLO Attainment:* 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, and
(3) *LSO Ablation Study:* QLM demonstrates that all LSOs contribute to SLO attainment and throughput improvement. Notably, we find that model warm start improves throughput by 300% in multi-model serving, and request eviction improves SLO attainment by 80% in single-model serving.

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 [21], Orca [57], TensorRT [48] and TGI [49] 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

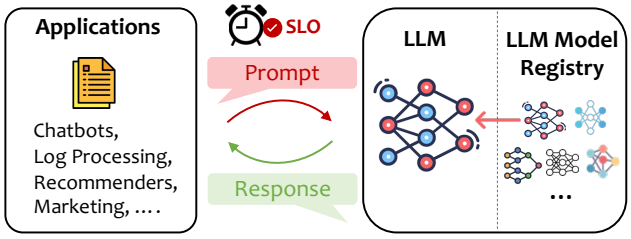

Figure 2: Interaction between LLM applications and a multi-model LLM serving system.

in significant memory waste as the KV cache grows dynamically during the decoding stage. PagedAttention [21] 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 [21].

2.2 LLM Serving Systems

Figure 2 shows the typical workflow of an LLM serving system. User-facing applications such as chatbots, log processing, and recommenders have specific end-to-end latency SLO requirements [4, 20, 37, 39]. 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 completion time <10s [14]), and batch jobs like log processing have a more relaxed SLO. Note that end-to-end SLOs do not specify per-token metrics such as time per output token (TPOT) or time to first token (TTFT) [10] as they cannot be directly translated to the application's latency requirements.

These requests are then served by an LLM serving system such as vLLM [21], TGI [49], and Nvidia Triton [51] that implement state-of-the-art inference optimizations such as continuous batching and paged attention. However, even though these systems are optimized for high throughput serving, served requests can suffer from *head-of-line (HOL) blocking* [38, 52] when the request arrival rate exceeds the LLM serving system throughput [53]. In such cases, *request queues would form, and requests with strict SLOs at the end of the queue would have to wait for earlier requests to complete, resulting in potential SLO violations.*

Managing these request queues is further complicated by the growth of specialized, fine-tuned models [18, 44] for various LLM-augmented applications. For example, Code Llama [42] is fine-tuned for coding assistance, and Llama-chat [50] is fine-tuned for chatbots. As maintaining a standalone LLM serving system for each of these models can be expensive, they often have to be multiplexed together sharing the same serving system [6, 22, 25, 44]. *However, serving multiple heterogeneous models with variable sizes results in high cold start time (that is needed for loading the model to the device memory) and low device utilization.*

LLM serving systems have to manage these requests and models with limited devices through various backend LLM serving operations (LSOs) as defined in Def. 2.4 such as request eviction [54], load balancing [15], and GPU-CPU state swap [46], in order to meet the SLO requirements. *Managing these LSOs can be complex because of (1) their inter-dependency with each other, and (2) the auto-regressive nature of LLMs where the length of output tokens is highly variable.*

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 *end-to-end time* required to complete the request (request waiting time and execution time) [30]. Note that SLOs in QLM are defined *per query* and not per token (e.g., TPOT or TTFT [10]). End-to-end request completion time is the prime metric valued by the customers using LLM inferencing services [30, 31] as it can be directly converted into downstream application SLOs.

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 [21] and TGI [49]. An LLM serving instance is composed of the LLM serving system and an LLM model that is being served.

Definition 2.4. *LLM Serving Operations (LSOs):* In each LLM serving instance, the models and requests in the queue are managed by the LLM serving operations (LSOs). Examples of LSOs include request eviction [54], load balancing [15], GPU-CPU state swap [46]. Each LSO can trigger an action that either depends on or alters the state of the request queue (see Section 5 for details).

2.4 Motivation and Characterization

To meet end-to-end application SLOs, it is critical to understand (1) the impact of LLM autoregressive patterns on the request completion time, (2) the complex interrelationship among individual LSOs, and (3) how end-to-end SLOs can be translated to actions for backend LSOs.

[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.

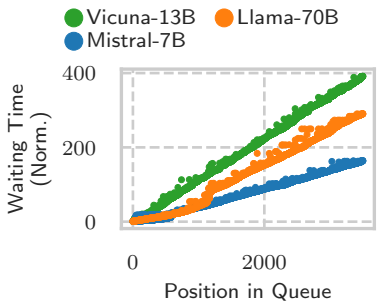

Figure 3: Requests have predictable waiting times in a continuous batching system.

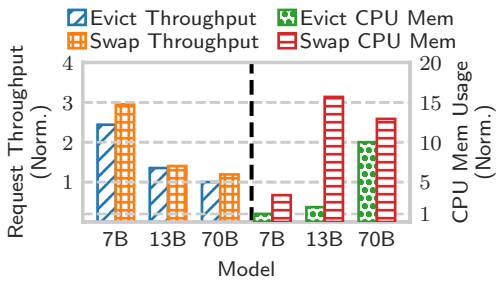

Figure 4: Choice between GPU-CPU state swapping and request eviction depends on the model setup.

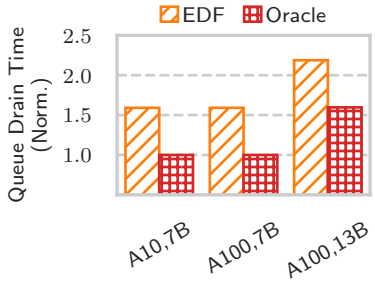

Figure 5: Model swapping and request pulling can jointly decrease queue drain time.

We characterize the performance of a state-of-the-art LLM serving system, vLLM [21], augmented with various LSOs to motivate the design of QLM. We use ShareGPT [43] and Azure LLM serving traces [36] from production environments. We present three key insights below.

Insight #1: ***Request waiting times can be accurately estimated with analytical methods.*** While individual request completion times in LLM inference can vary significantly, the average waiting time for a request in the queue is predictable. The waiting time can be estimated by dividing the total number of output tokens for requests ahead in the queue with the token generation throughput. Both average number of output tokens and throughput can be estimated by profiling the workload over time. We empirically find that these statistics do not change significantly for a workload and hence can be assumed constant. We validate the above waiting time estimation formula using real-world traces [36, 43]. 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). In Section 6, we extend this estimator further to support request completion times estimation.

Insight #2: ***Selecting the optimal LSO depends on the model and hardware setup.*** When multiple LSOs are enabled for LLM inference, there can be conflicts between the goals of two LSOs. For example, when a request is blocked in the queue due to out-of-capacity on the GPU device serving the running batch, two LSOs can be used to allow immediate execution: request eviction and GPU-CPU state swapping. This choice primarily depends upon the trade-off between available CPU memory and the model/hardware-dependent computation cost of swapping vs. eviction. If additional memory consumption per KV cache token is lower relative to the KV recomputation time, then GPU-CPU state swapping would become preferable (and vice-versa).

To demonstrate this trade-off, we perform experiments with varying-sized models on NVIDIA A100 GPUs. Figure 4 shows the request throughput and CPU memory usage across these experiments. For models such as Mistral-7B, swapping increases request throughput by ∼20% with a relatively small CPU memory overhead of 20 GB. On the other hand, for models such as Vicuna-13B, swapping increases CPU memory requirements by 8.4× (from 26 GB to 220 GB) with negligible throughput benefit. Therefore, the optimal LSO selection in this example would be to prefer eviction for Vicuna-13B and state swapping for Mistral-7B.

Insight #3: ***Multiple LSOs can work together to attain SLOs while improving serving throughput.*** While some LSOs can have conflicting actuation actions (as shown in Insight #2), others, such as model swapping and request pulling, can work together to attain end-to-end latency SLOs while improving the throughput of the LLM serving system. Consider the case where each LSO is being optimized independently. The optimal request pulling strategy is to use the Earliest Deadline First (EDF) scheduling to achieve the maximum number of requests that meet their SLOs. *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 the end-to-end latency SLOs by providing a translation layer between the SLOs and the actions for the LLM Serving Operations (LSOs).

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 [40] that provides the requisite fault tolerance and consistency properties.

Formation of Request Groups. Each 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. 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 LSOs. 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 virtual queues 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 completion time (RCT) estimator* (see Section 6) and *plan generator* (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.

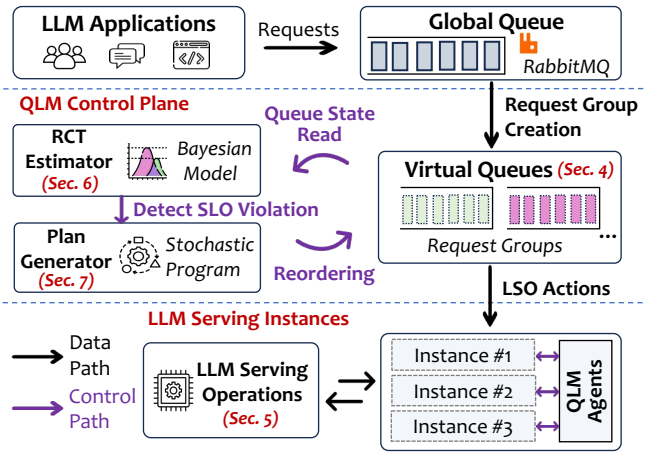

Figure 6: Overview of QLM.

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* (see Section 4) to reduce the input space of the optimization solvers, thus lowering the computational overhead and enabling scalability.

Design Principle #2: ***Handling a diverse set of LSOs with complex inter-relationships.*** In Section 2.4 (Insight #2 and Insight #3), we demonstrate that multiple LSOs can either conflict with each other or have a synergistic effect to enhance overall serving performance. To attain model-serving latency SLOs, it is thus critical to translate an end-to-end SLO to the appropriate backend LSO actions. QLM models these complex inter-relationships with a two-step approach. First, *virtual queues* (see Section 4) enable the necessary abstraction to enable actions for multiple backend LSOs. Second, the *plan generator* (see Section 7) models the impact of ordering on multiple LSOs with a stochastic programming solver. We specifically prefer a stochastic programming solver over other optimization methods because it: (1) allows us to represent the stochastic nature of LLM's autoregressive generation, and (2) systematically considers various constraints introduced by multiple LSOs.

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 plan generator has to consider each device's computing power, memory capacity, and memory bandwidth. The *RCT estimator* (see Section 6) estimates this impact of heterogeneity for the plan generator. The profiling costs for the RCT 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

```
groups ← kMeansClustering(requests)
for i ← 1 to length(groups) do
    if groups[i].size() > avg_batch_size × δ then
        newGroups ← groups[i].splitHalf()
        groups.append(newGroups)
    end if
end for
```

4 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. QLM's virtual queues draw inspiration from virtual output queuing (VOQ) [32, 47], a popular architecture used in network switches to address head-of-line blocking.

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 RCT 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 end-to-end 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 (δ) 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 plan generation; (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.

Handling New Incoming Requests. As new requests join the global queue, they are classified into the existing request groups, and the RCT estimator calculation (Section 6) is triggered to find out whether any SLOs are being violated. Upon any SLO violation, the plan generator (Section 7) 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 [40] 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 plan generator. (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 plan generator (as described in Section 7).

Fig. 7 shows the five 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 or GPU-CPU state swap. Model warm start 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.

Request Pulling (❶ 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

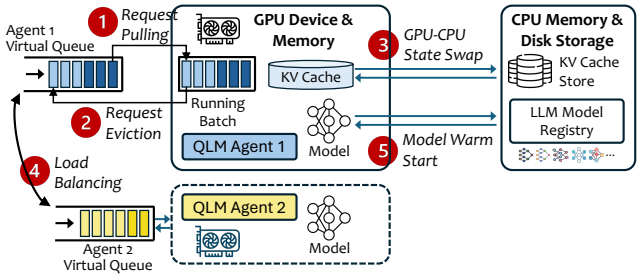

Figure 7: Basic LLM serving operations (LSOs) for an LLM-serving instance that a QLM agent manages.

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 RCT estimator detects an SLO violation, and the plan generator 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. As each request is evicted, it loses its KV cache, which needs to be regenerated in the future. However, the regeneration process is faster than the original prompt generation as the output tokens generated so far are appended to the input prompt to generate the remaining output tokens. Therefore, the original prefill and multiple decode steps are replaced by a single prefill step.

GPU-CPU State Swapping (3 in Fig. 7). Request eviction requires repeating the prefill step when re-executing the evicted requests, leading to repetitive and redundant computation. The prefill step could be costly for less powerful GPU hardware or complicated model types. An alternative to request eviction is to migrate the KV cache of evicted requests into the CPU memory. 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. As an alternative to request eviction, GPU-CPU state swapping can also be invoked when the head request group is updated.

Load Balancing (4 in Fig. 7). As each LLM serving instance is associated with a separate virtual queue, the plan generator'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 Warm Start (5 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 state 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 only saves the output tokens generated, and for state swapping, it 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 Completion Time (RCT) Estimator

For requests within each request group, the RCT estimator leverages a Bayesian statistical approach to generate probability distributions of the completion times and waiting times. To create such RCT estimator, we extend the estimator described in Section 2.4 further by accounting for the variance per request in completion times. The process is explained below with variable definitions listed in Table 1.

Table 1: Glossary of symbols used in the RCT 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
Θ	Token generation throughput
ε	Inefficiency factor due to continuous batching
d	Decode time per output token

As shown in Equation 1, the total 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 as it is a highly parallel GPU-accelerated operation whose time increases minimally as the number of input tokens increases. 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. As token generation throughput has a low variance due to continuous batching (characterized in Insight #1 Section 2.4), we consider the token generation throughput (Θ) to be constant throughout the token generation process. 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 (Θ) where i denotes each of the $q-1$ requests in the queue ahead of the request we model.

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

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 normal distribution (as shown in Equation 3) with the mean μ_o and standard deviation σ_o fitted from the offline dataset of the request input-output history for the request group that the request q belongs to.

$$O_q \sim N(\mu_o, \sigma_o) \tag{3}$$

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

$$D_q = O_q \times \varepsilon \times d \tag{4}$$

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 (due to iteration-level scheduling) 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 ε, 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.

$$C = \max_q C_q \tag{5}$$

Offline Profiling. There are two independent profiling steps required for the RCT 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, such as the prefill time (P), decode time per iteration (d), and inefficiency factor (ε), are obtained during hardware profiling by logging metrics from the LLM serving instance.

Table 2: Glossary of symbols used in the stochastic 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
$ct_{g,j}$	Request group completion 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

7 Plan Generator

The plan generator is invoked by the RCT estimator when an SLO violation is likely to occur. Upon invocation, the plan generator runs a stochastic programming model to reorder the virtual queues that decide underlying LSO actions to maximize SLO attainment. Details on how the queue ordering leads to the selection and actuation of LSO actions in each LLM serving instance are defined in Section 5. The plan generator uses a stochastic 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 stochastic programming model with its defined variables listed in Table 2.

Overall Modeling Approach. The goal of the stochastic 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 completion time and SLO value. If SLOs are met, all penalty terms would be smaller than 0. Given the SLOs as the inputs to the stochastic programming model, we obtain the request group completion time estimation from the RCT estimator (see Section 6) to estimate the defined penalty terms. The completion time for a request group is the sum of the waiting time for request groups ahead in the virtual queue (from Equation 2), the completion time for the request group (from Equation 5), and swap times associated with transferring model weights into GPU memory. [2] 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 RCT estimator profiling.

Mathematical Definitions of Constraints. Now, we describe each of the constraints mathematically 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.

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

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 it further [7].

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

The cumulative completion 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. We assume a constant conservative estimate of swap time, but this can be further improved in future work.

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

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

Note that completion times are a stochastic variable as group completion times $C_{g,i}$ are represented with a probability distribution (i.e., C from Equation 5).

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

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

The final constraint is that all penalty values should be less than 0 (i.e., all SLOs are satisfied) with a high probability.

$$P(p_{g,j} \leq 0) > \Delta \forall g, j \tag{12}$$

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

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

8 Evaluation

Our experiments address the following research questions:

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

Experiment Setup. We evaluate QLM on multiple varying-sized open-source LLMs: Mistral-7B [19], Vicuna-13B [5], and Llama-70B [50]. Our test cluster consists of 80–100 GPUs of two types: NVIDIA A10 (24 GB memory) and 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* [21]: Requests use the default first-come-first-serve (FCFS) scheduler in vLLM, and (3) *SHEPHERD* [60]: Requests use 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.

Workloads. We consider the following four workloads that are derived from the requirements of a production cloud service provider: **[W_A] (*Multi-Model*)** Requests to different models arrive at the queue, and the number of served models is larger than the number of corresponding LLM serving instances. Consequently, more than one model needs to be

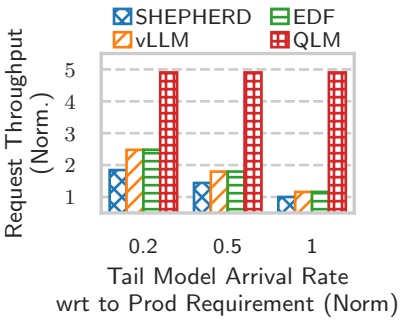

Figure 8: Multi-model request serving throughput .

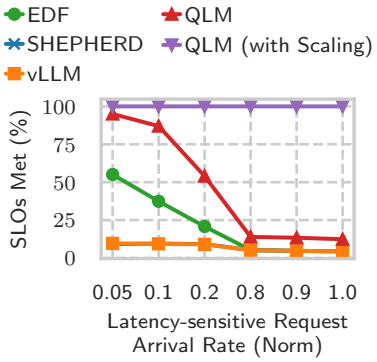

Figure 9: Multi-model SLO satisfaction.

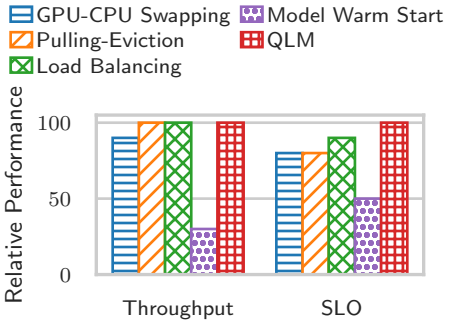

Figure 10: Multi-model LSO ablation study.

multiplexed on the same LLM serving instance. Additionally, based on our insights from production request patterns, we assume that $> 50\%$ of models are "tail models", i.e., they are not used often and only occasionally need to be swapped in place of the frequently used models. Each request type has varied SLO requirements based on the downstream application requirements. For example, latency-sensitive services have a completion time SLO <30s, and batch jobs have SLOs in the order of several minutes. While our choice of SLO values is motivated by production requirements, we also provide robustness analysis for alternative SLO values in Appendix A.1. **[W_B] (*Single-Model*)** Requests to a single model arrive at the request queue, and no model swapping is required. Similar to W_A, these requests have multiple SLOs dependent on the underlying application. **[W_C] (*MegaPrompt*)** The request workload consists of several "mega prompts" in addition to the workload from W_A. These mega prompts have a large number of input and output tokens that occupy a large percentage of GPU memory and potentially prevent requests in the waiting queue from executing.

8.1 Multi-Model Evaluation

We run workload W_A to evaluate the *multi-model* LLM serving performance on A100 GPUs with respect to request throughput and end-to-end latency 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 8 shows the request throughput (i.e., requests served per second) for W_A comparing QLM with the baseline mechanisms for varying percentage of tail model arrival rates (normalized to peak tail model demand). 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 plan generator couples every tail model with another frequently accessed 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 9 shows the percentage of SLO satisfied for the latency-sensitive services against the percentage of their occurrence in the whole serving workload. When these latency-sensitive services constitute less than 5% of the request queue, QLM satisfies more than 90% of all SLO values. As the relative percentage of latency-sensitive service requests increases, no combination of requests would be able to meet all SLOs, and the plan generator 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 8.2.

Contribution of Each LSO. Each of the five LSOs used by QLM, including request pulling/eviction, GPU-CPU state swapping, model warm start, and load balancing, contributes to either the latency and/or the throughput of the serving system. Figure 10 shows the impact of removing each LSO on QLM performance for W_A. 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.

8.2 Single-Model Evaluation

We run workload W_B on 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 8.1).

Request Throughput and SLO Attainment. Figure 12 shows the percentage of SLOs that are satisfied by QLM and the baseline systems. Similar to the multi-model case, we find that when the queue primarily consists of latency-sensitive services, none of the systems can satisfy the SLOs. This is because the minimum serving time is much longer

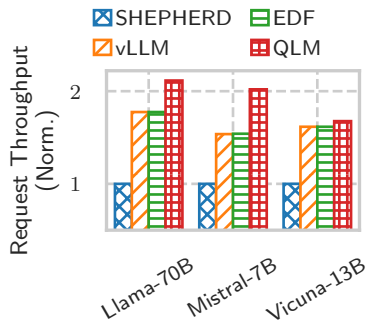

Figure 11: Single model request serving throughput.

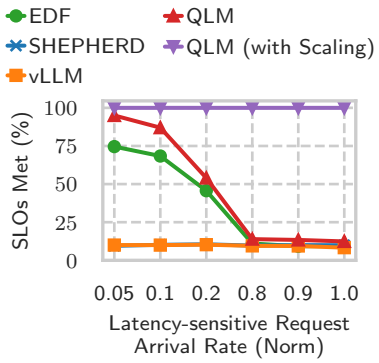

Figure 12: Single model SLO satisfaction.

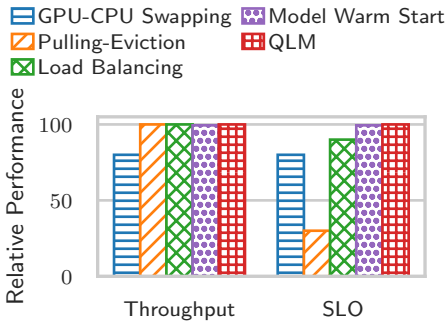

Figure 13: Single model LSO ablation study.

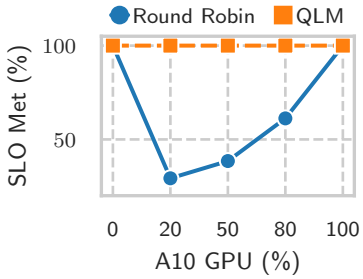

Figure 14: Impact of hardware heterogeneity.

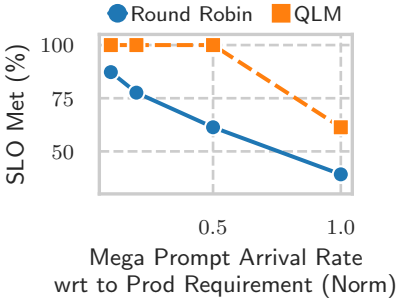

Figure 15: Impact of mega prompt arrivals.

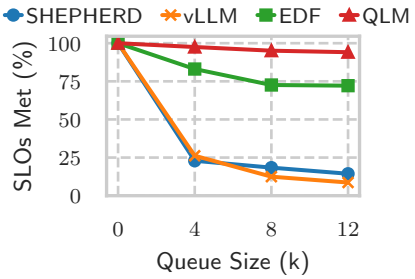

Figure 16: Impact of increasing queue size on SLO satisfaction.

than the specified SLO. As the number of latency-sensitive service 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 latency-sensitive service requests ahead in the queue, (b) Compared to EDF, QLM enables appropriate selection between GPU-CPU state swapping and request eviction LSOs, and (c) Compared to SHEPHERD, QLM uses continuous batching as opposed to static batch size and models the auto-regressive LLM nature with the RCT 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 11 shows the request throughput for QLM and the individual baseline mechanisms. QLM achieves higher throughput, i.e., 20% higher compared to vLLM and EDF, and 50% higher than SHEPHERD.

Contribution of Each LSO. Figure 13 shows the impact of removing each LSO considered by the backend LLM serving instance in QLM. Scheduling and request eviction contribute significantly to latency reduction for latency-sensitive services and consequently increase the number of SLOs met. GPU-CPU state swap 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.

8.3 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 14 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 RCT estimator with offline profiling, and the plan generator 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 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).

Mega Prompt Workload. The RCT estimator of QLM takes into account input and output token distribution when estimating the request completion time. Consequently, when there

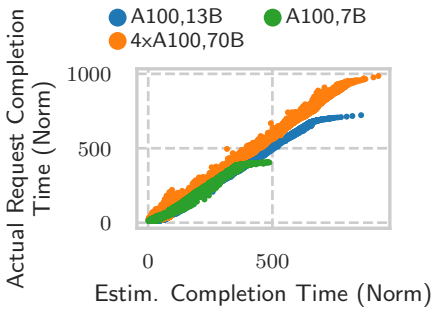

Figure 17: Accuracy of RCT estimator.

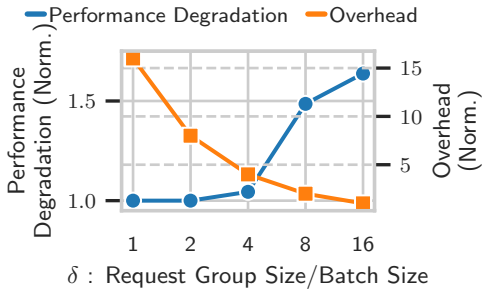

Figure 18: Impact of request group size on QLM performance.

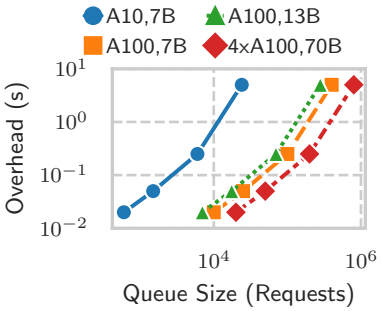

Figure 19: QLM Overhead.

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 15 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 16. 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 8.2, while QLM is able to maintain a high SLO satisfaction percentage.

RCT Estimator Accuracy. The RCT estimator calculates the request completion time based on initial profiling of the model and hardware setup. This initial profiling time is negligible as only a single batch of requests need to be run on the GPU. As described in Section 6, QLM does not assume that the exact output tokens are known ahead of time, but instead uses the workload output token distribution. Figure 17 shows the estimated request completion time vs. the actual request completion time for the RCT estimator across different models and hardware configurations. Overall, we find that the RCT estimator has a high accuracy in estimating request completion times with an R^2 (coefficient of determination) value of 0.99 (out of 1.0). While the RCT estimator is highly accurate in estimating request completion time, it is not perfect. There could be requests with an outlier number of output tokens, leading to underestimation and potential SLO violations. However, current LLM serving systems typically have a hard limit on the maximum number of output tokens [21, 36], which eliminates the presence of these outliers.

Impact of Request Group Size. QLM sets the request group size as a multiple (δ) of the average batch size. The exact δ value depends on the acceptable trade-off between the overhead of running the plan generator and the granularity of decision-making. As δ becomes smaller, QLM achieves a finer granularity of decision-making, leading to improved performance. However, the overhead leads to delayed decision-making. Figure 18 demonstrates this tradeoff between performance degradation (caused by changing granularity in decision making) and overhead of the plan generator when varying δ. 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 overhead of QLM largely depends on the time required to solve the stochastic programming formulation required by the plan generation. In Figure 19, we show the time required to solve for the plan generator 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).

9 Related Work

General ML Model-Serving Systems. Traditional model-serving systems provide functionalities such as scheduling, placement, batching, and autoscaling. Clipper [9], TensorFlow-Serving [35], MArk [59], InferLine [8], Shepherd [60], and Clockwork [16] are some earlier work on serving traditional ML models like ResNet that are relatively small. INFaaS [41] and Cocktail [17] 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.

LLM Scheduling Optimization. Existing state-of-the-art LLM serving systems [21,49,51,57] 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 [55] proposes preemptive scheduling with a Multi-Level Feedback Queue. Andes [27] defines Quality-of-Experience (QoE) for LLM serving as token delivery speed, and proposes a preemptive scheduler that maximizes QoE. QLM is the first queue management framework that optimizes end-to-end latency 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, 11, 13, 23, 26, 28, 29, 33, 36, 44, 45, 61, 63, 64]. 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).

10 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%. We further discuss extensions to QLM in Appendix A.3.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. APIServe: Efficient API support for large-language model inferencing. *arXiv preprint arXiv:2402.01869*, 2024.

[2] 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. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[4] Shuang Chen, Christina Delimitrou, and José F Martínez. 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)*, pages 107–120, 2019.

[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. Vicuna: An open-source chatbot impressing GPT-4 with 90% ChatGPT quality, March 2023.

[6] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 2022)*, pages 199–216, 2022.

[7] Marco Cococcioni and Lorenzo Fiaschi. The big-m method with the numerical infinite m. *Optimization Letters*, 15(7):2455–2468, 2021.

[8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. InferLine: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*, pages 613–627, 2017.

[10] LLM inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices. Accessed: 2024/04/10.

[11] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 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)*, pages 389–402, 2021.

[12] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Locality-enhanced serverless inference for large language models, 2024.

[13] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for LLMs, 2024.

[14] Ulrich Gnewuch, Stefan Morana, Marc TP Adam, and Alexander Maedche. Opposing effects of response time in human–chatbot interaction: the moderating role of prior experience. *Business & Information Systems Engineering*, 64(6):773–791, 2022.

[15] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. Mélange: Cost efficient large language model serving by exploiting GPU heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

[16] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 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)*, pages 443–462, 2020.

[17] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2022)*, pages 1041–1057, 2022.

[18] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[19] 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. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

[20] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated SLO for enterprise clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, pages 117–134, 2016.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023)*, pages 611–626, 2023.

[22] Matthew LeMay, Shijian Li, and Tian Guo. Perseus: Characterizing performance and cost of multi-tenant serving for CNN models. In *2020 IEEE International Conference on Cloud Engineering (IC2E 2020)*, pages 66–72. IEEE, 2020.

[23] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*, pages 19274–19286. PMLR, 2023.

[24] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. CaraServe: CPU-assisted and rank-aware LoRA serving for generative LLM inference. *arXiv preprint arXiv:2401.11240*, 2024.

[25] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 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)*, pages 663–679, 2023.

[26] Pierre Lienhart. 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).

[27] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and enhancing quality-of-experience in LLM-based text streaming services. *arXiv preprint arXiv:2404.16283*, 2024.

[28] Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time, 2023.

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

[30] Azure LLM inference services. https://learn.microsoft.com/en-us/azure/machine-learning/concept-endpoints?view=azureml-api-2. Accessed: 2024/04/10.

[31] Google LLM inference services. https://cloud.google.com/dialogflow/pricing#dialogflow-pricing. Accessed: 2024/04/10.

[32] Nick McKeown, Martin Izzard, Adisak Mekkittikul, William Ellersick, and Mark Horowitz. Tiny Tera: a packet switch core. *IEEE Micro*, 17(1):26–33, 1997.

[33] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. SpotServe: Serving generative large language models on preemptible instances. *arXiv preprint arXiv:2311.15566*, 2023.

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

[35] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. TensorFlow-Serving: Flexible, high-performance ML serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

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

[37] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 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)*, 2020.

[38] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Efficient interactive LLM serving with proxy model-based sequence length prediction. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, volume 5, pages 1–7, San Diego, CA, USA, 2024. Association for Computing Machinery.

[39] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Power-aware deep learning model serving with μ-serve. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 2024)*, 2024.

[40] RabbitMQ. https://www.rabbitmq.com/.

[41] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *Proceedings of 2021 USENIX Annual Technical Conference (ATC 2021)*, pages 397–411, 2021.

[42] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[43] ShareGPT dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

[44] 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. S-LoRA: Serving thousands of concurrent LoRA adapters, 2023.

[45] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 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)*, pages 31094–31116. PMLR, 2023.

[46] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. DéjàVu: KV-cache streaming for fast, fault-tolerant generative LLM serving, 2024.

[47] Yuval Tamir and Gregory L Frazier. High-performance multi-queue buffers for VLSI communications switches. *ACM SIGARCH Computer Architecture News*, 16(2):343–354, 1988.

[48] TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM. Accessed: 2024-04-10.

[49] Text Generation Inference. https://github.com/huggingface/text-generation-inference. Accessed: 2024-04-10.

[50] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[51] Nvidia Triton Inference Server. https://developer.nvidia.com/triton-inference-server. Accessed: 2024-04-10.

[52] Tobias Viernickel, Alexander Froemmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. Multipath QUIC: A deployable multipath transport protocol. In *2018 IEEE International Conference on Communications (ICC 2018)*, pages 1–7. IEEE, 2018.

[53] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Towards efficient and reliable LLM serving: A real-world workload study, 2024.

[54] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. LoongServe: Efficiently serving long-context large language models with elastic sequence parallelism. *arXiv preprint arXiv:2404.09526*, 2024.

[55] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[56] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 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)*, pages 521–538, 2022.

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

[59] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC 2019)*, pages 1049–1062, 2019.

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

[61] 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. H_2O: Heavy-hitter oracle for efficient generative inference of large language models, 2023.

[62] Juntao Zhao, Borui Wan, Yanghua Peng, Haibin Lin, and Chuan Wu. LLM-PQ: Serving LLM on heterogeneous clusters with phase-aware partition and adaptive quantization. *arXiv preprint arXiv:2403.01136*, 2024.

[63] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024.

[64] Lei Zhu, Xinjiang Wang, Wayne Zhang, and Rynson W. H. Lau. RelayAttention for efficient large language model serving with long system prompts, 2024.

A Appendix

A.1 Sensitivity Analysis on SLO Values

In Section 8, we configure SLO values that are derived from our production system requirements. To further explore the choice of SLO value on serving performance, we conduct a sensitivity study by varying the SLO values and showing the impact on the performance of QLM and other baselines for W_A. Figure 20 shows the impact of increasing the SLO scale (i.e., the ratio of SLO value to original SLO requirement) on SLO satisfaction across requests. As SLOs are relaxed, more baselines are able to meet all SLOs. For example, Earliest Deadline First (EDF) and SHPEHERD are the first to be able to meet all SLOs, when the SLO value is relaxed to 3× and 5× the original SLO value, respectively. This is expected as both of these mechanisms have an inherent notion of request ordering based on SLO values. When the SLO value is relaxed by 11×, vLLM is also to meet all SLOs. vLLM requires relatively high SLO relaxation because it adopts a simple first-come-first-serve (FCFS) scheduling policy without any notion of SLO-based ordering, which results in much worse HOL blocking compared to the other approaches.

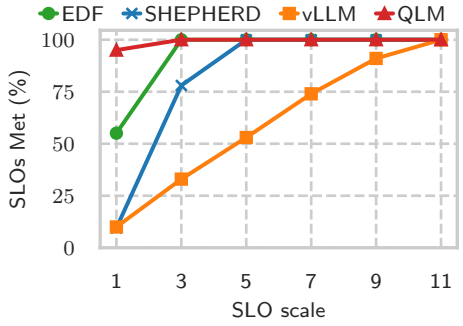

Figure 20: Impact of varying SLO values on QLM.

A.2 GPU Utilization

QLM achieves higher SLO attainment and equal or higher throughput compared to other baseline approaches with the same number of GPU devices. Consequently, the utilization of these resources is expected to be higher. To demonstrate the improvement in resource utilization, we measure the average GPU memory utilization over time. GPU memory utilization is specifically chosen as LLM serving is primarily a memory-bound workload. Figure 21 shows the GPU memory utilization for W_A (multi-model) and W_B (single model) workloads. For W_A, we find that GPU utilization is significantly higher as QLM avoids repeated model swaps that keep the memory utilization low. For W_B, we find that GPU memory utilization is similar between QLM, EDF, and vLLM as all mechanisms use continuous batching to improve device memory utilization, and no model swapping is needed for the single-model setup. On the other hand, SHEPHERD uses dynamic batching and does not saturate GPU memory, leading to lower memory utilization and throughput.

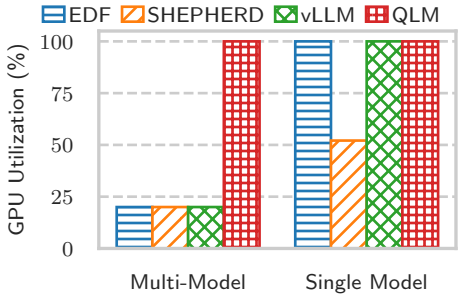

Figure 21: GPU memory utilization comparison across LLM serving systems under multi-model and single-model setups.

A.3 Discussion

What happens if QLM is unable to meet SLOs? QLM's plan generator 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 9 and Figure 12, (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 [34]) can be added as an LSO with additional constraints on memory in the stochastic 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 $priority(request_1) < priority(request_2)$. With strict priority, the relative ordering of requests across priorities is pre-decided, however per-priority assignment still needs to be optimized to minimize request completion times. Consequently, the concepts of virtual queues, request groups, and the RCT estimator continue to remain useful. However, the underlying constraints of the stochastic programming model in the plan generator would need to be updated to support constraints unique to priorities.

Can SLOs be defined on just the waiting time? QLM addresses the problem of optimizing end-to-end latency SLO

attainment. However, it can also be made to work with the modified objective of waiting time SLOs (i.e., excluding the execution time). In this case, the RCT estimator and plan generator can be simplified as only the waiting time (W_q) needs to be considered for each request group. Additionally, we can substitute the stochastic programming formulation with a linear programming solver as waiting times are largely deterministic, as shown in Insight #1 Section 2.4.