SeaLLM: Service-Aware and Latency-Optimized Resource Sharing for Large Language Model Inference

Yihao Zhao
Peking University

Jiadun Chen
Huawei Cloud

Peng Sun
Shanghai AI Lab

Lei Li
Peking University

Xuanzhe Liu
Peking University

Xin Jin
Peking University

Abstract

Large language models (LLMs) with different architectures and sizes have been developed. Serving each LLM with dedicated GPUs leads to resource waste and service inefficiency due to the varying demand of LLM requests. A common practice is to share multiple LLMs. However, existing sharing systems either do not consider the autoregressive pattern of LLM services, or only focus on improving the throughput, which impairs the sharing performance, especially the serving latency. We present SeaLLM, which enables service-aware and latency-optimized LLM sharing. SeaLLM improves the overall sharing performance by (1) a latency-optimized scheduling algorithm utilizing the characteristics of LLM services, (2) a placement algorithm to determine the placement plan and an adaptive replacement algorithm to decide the replacement interval, and (3) a unified key-value cache to share GPU memory among LLM services efficiently. Our evaluation under real-world traces and LLM services demonstrates that SeaLLM improves the normalized latency by up to 13.60×, the tail latency by up to 18.69×, and the SLO attainment by up to 3.64× compared to existing solutions.

1 Introduction

The fast development of large language models (LLMs) is changing modern applications, including chatbot [16], code generation [31], text process [13], and embodied AI [41]. Many organizations [8, 13, 19, 54] have proposed their foundation LLMs with different architectures and model sizes. Service providers usually deploy different LLMs for specific tasks [3, 18, 51]. However, LLM services have a higher demand for GPU resources in computation, memory, and communication than classical deep learning (DL) models. The service providers need clusters with high-performance GPUs to provide high-quality LLM services, leading to high costs in building, using, and maintaining the clusters.

The request traffic of LLM services is dynamic for different time and services according to the data from production LLM services [42, 48] as shown in Figure 1. Therefore, serving each LLM with dedicated GPUs leads to substantial wastes of expensive GPU resources and impacts the serving performance [18, 32]. Some systems [4, 18, 22, 32] share resources to improve serving performance and resource utilization.

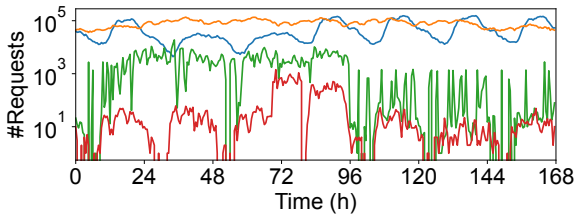


Figure 1: The request traffic of four production LLM services during seven days [42, 48].

Most existing sharing systems [4, 22, 32] are designed for serving classical DL models like the convolutional neural network (CNN) and they usually use the first-come-first-serve (FCFS) scheduling algorithm. However, LLM services have a unique autoregressive pattern, i.e., generating the output tokens one by one. The autoregressive pattern results in various output lengths and serving time for different LLM services and requests. Therefore, the FCFS scheduling algorithm faces the head-of-line blocking problem for the shared LLM services, where a long request can block the following requests and affect the overall serving performance [27, 50].

Recently, MuxServe [18] is proposed for sharing LLM services. MuxServe utilizes spatial and temporal multiplexing to improve the system throughput. However, it influences the request latency, which is important to LLM services [56]. The reasons are mainly two-fold. First, MuxServe adopts the round-robin (RR) scheduling algorithm for the prefill phase and the FCFS scheduling algorithm for each service. The RR scheduling algorithm can avoid the head-of-line blocking problem [10, 14], but is still not adequate to serve multiple LLM services. For long and bursty requests, the RR scheduling algorithm introduces a heavy burden on memory because LLM serving systems usually use the key-value cache (KV cache) to speed up serving and it requires storing

the key-value pairs (KV pairs) of all requests. Besides, the RR scheduling algorithm increases the latency for bursty requests. Second, MuxServe predefines a quota of computation resources for each LLM service to enable spatial multiplexing. The predefined quota cannot react to the dynamic request traffic timely, leading to GPU waste and an increase in latency.

This paper presents SeaLLM to support service-aware and latency-optimized resource sharing for LLM services. The key observation is that LLM services usually have their specific characteristics, including the iteration time and the distributions of input and output lengths. Based on the characteristics, we place the LLM services and schedule the requests to alleviate the head-of-line blocking problem caused by FCFS and the burden of GPU memory caused by RR.

SeaLLM exploits a service-aware and latency-optimized scheduling algorithm for the shared LLM services. We assign a priority based on the executed time and the service characteristics to each request. Then we schedule the requests preemptively according to the priority. The scheduling algorithm is proved to minimize the normalized latency with accurate profiling information. However, in practice, some requests may deviate from the service characteristics and break the optimality of the scheduling algorithm. Intuitively, for the overall performance, if one request exceeds the estimated execution time, its priority should be decreased to allow other requests to be processed. With this intuition, we propose a doubling budget (DB) scheduling algorithm, which assigns a budget of execution time based on the profiled characteristics of each LLM service. The DB scheduling algorithm gradually decreases the priority of one request when its budget is run out and doubles the budget to continue serving the request.

SeaLLM leverages a search-based placement algorithm to generate the placement plan, i.e., how to place the LLM services in a cluster. The time complexity of finding the optimal placement plan grows exponentially along with the cluster size and the number of LLM services. To address this problem, we separate the search process into two stages, i.e., GPU group partition and LLM service allocation. We also propose several heuristics to reduce the search time further. With parallel execution, we can decrease the search time to minutes and fully hide the search process with service execution. Besides, due to the time-varying request traffic, a fixed replacement is not a panacea. Commonly, the solution to this problem is to replace the placement periodically [52, 55]. However, prior methods leverage a fixed replacement interval, which is hard to decide. SeaLLM adopts an adaptive replacement algorithm, which changes the replacement interval according to the estimated performance and the real-achieved performance.

We implement SeaLLM as a cluster system with both a cluster manager to orchestrate the LLM services and requests, and local engines to execute the shared LLM services. For the shared LLMs, memory management is a knotty problem, especially the dynamic KV cache. A cluster system should share the KV cache of different LLM services to improve the serving performance [18]. However, the cache block shape is determined by the LLM architecture and is not identical for all LLMs, making it difficult to share the KV cache. SeaLLM utilizes a unified KV cache with merged cache blocks to balance the block table size, locality, and fragmentation.

We evaluate SeaLLM on a 32-GPU cluster with real-world traces [39] and widely-used LLMs [44, 54]. The experiment results show that SeaLLM improves the normalized latency by up to 13.60×, the tail latency by up to 18.69×, and the service level objective (SLO) attainment by up to 3.64× compared to existing state-of-the-art (SOTA) systems. At the token level, SeaLLM improves the average time-to-first-token (TTFT) by up to 51.98× and reaches a similar average time-per-output-token (TPOT) with SOTA systems.

In summary, this paper makes the following contributions:

- We identify the limitations of existing sharing solutions and propose to utilize the characteristics of LLM services for service placement and request scheduling.
- We propose a latency-optimized scheduling algorithm utilizing the service characteristics for shared LLM services.
- We propose a placement algorithm and an adaptive replacement algorithm to determine the placement plan.
- We conduct a comprehensive evaluation of SeaLLM with real-world traces and LLM services.

2 Background and Motivation

2.1 LLM Service

LLMs have been widely deployed for different tasks, such as chatbot, search engine, text process, etc [13, 16]. In practice, the foundation LLMs are usually finetuned with domain-specific data to serve specific tasks [51]. The LLM developed and deployed for a specific task is called an LLM service.

LLMs generate the output in an *autoregressive pattern*, i.e., the LLM generates one token in one step and the process is repeated until meeting the termination conditions. Naturally, the generation process is separated into two phases: prefill phase and decoding phase. The prefill phase processes all the input tokens concurrently and gets the first output token. The decoding phase only generates one output token in each step. To avoid the redundant computation brought by the attention layers, LLMs usually adopt *KV cache* [37] to cache the intermediate data, i.e., the KV pairs, in the GPU memory. The KV cache is formed by multiple cache blocks and the block shape is decided by the number of attention layers, the number of attention heads, and the hidden sizes. To conclude, LLM services have two main characteristics. First, the serving latency varies with different model architectures, input lengths, and output lengths. Second, the serving process requires large amounts of GPU memory to store the KV pairs.

LLM serving systems usually aim to reduce the serving latency, improve the SLO attainment, and improve the serving throughput. To achieve these goals, existing LLM serving

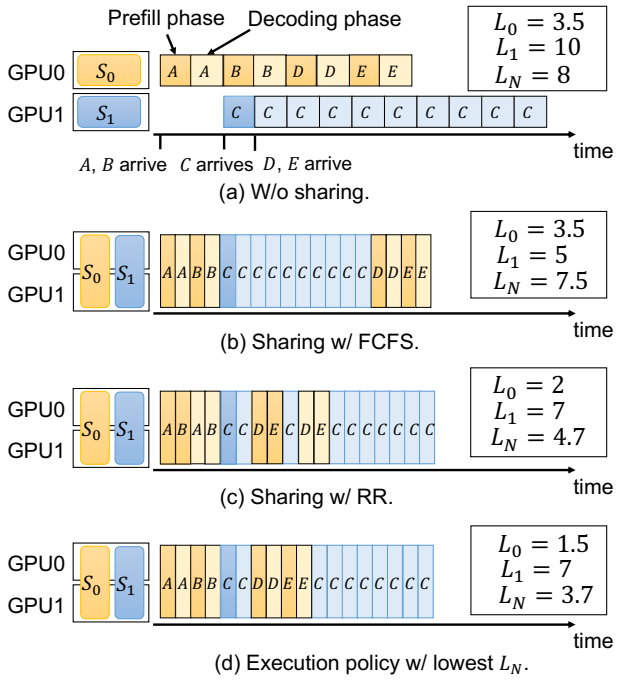


Figure 2: Four execution strategies of serving two LLM services on two GPUs. The blocks with dark colour are of the prefill phase and the blocks with light colour are of the decoding phase. L_0 and L_1 represent the average latency of service S_0 and S_1, respectively. L_N represents the normalized latency of S_0 and S_1, defined in Equation 1.

systems leverage techniques including kernel-level optimization [11, 17, 24], batching [20, 53], disaggregation [56], model parallelism [37, 53], etc. These techniques improve the performance of a single LLM service and they are orthogonal to the cluster setting which contains multiple LLM services.

2.2 LLM Sharing Systems

Figure 1 shows the dynamic nature of LLM services. The data are from two public traces of four production LLM services [42, 48]. The request traffic of LLM services is time-varying and different services show different popularities. The dynamic nature of LLM services brings opportunities for sharing LLM services to improve resource utilization and reduce costs for LLM service providers [18, 32]. Additionally, resource sharing can also benefit end users by improving the serving performance. The example in Figure 2 further illustrates the benefits of sharing LLM services. For two LLM services (S_0 and S_1) and five requests ($A-E$), the execution strategy without sharing has many blanks, indicating the waste of GPU resources. To better utilize the GPUs, we can collocate S_0 and S_1 on both GPUs by tensor parallelism as shown in Figure 2(b)-(d). The average latencies of both services (L_0 for service S_0 and L_1 for service S_1) are decreased.

Most existing sharing systems [22, 32] are designed for classical models. These models do not use the autoregressive pattern, i.e., for each request, the inference is only performed once. Previous systems utilize the FCFS scheduling algorithm which has the lowest scheduling overhead. However, as shown in Figure 2(b), the FCFS scheduling algorithm has the head-of-line blocking problem when applied to LLM services. Specifically, the long request C blocks the following requests D and E, and influence the average latency of S_0. As a result, the L_0 of FCFS is 2.3× longer than the L_0 of (d).

Service	Dataset	Avg input length	Avg output length
Chatbot	ShareGPT [5]	73.0	426.9
Summarization	LongBench [12]	13186.8	21.1
Code	HumanEval [15]	156.5	66.9

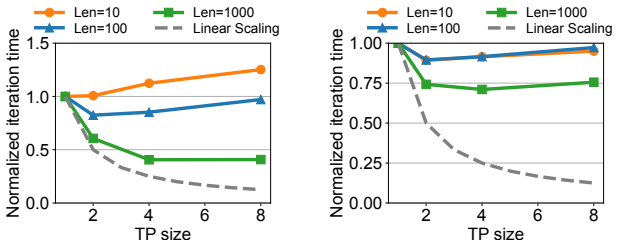
Table 1: The average input length and output length of different LLM services.

Recently, MuxServe [18] is proposed for sharing LLM services. It utilizes the RR scheduling algorithm to select services for the prefill phase, and uses spatial multiplexing to share the decoding phase. These mechanisms alleviate the problem of the FCFS scheduling algorithm. However, it still influences the serving performance, especially the latency, because of two reasons. First, the RR scheduling algorithm increases the request latency and the memory requirements for bursty requests. As shown in Figure 2, (c) RR has a larger average latency L_0 than (d), because requests D and E are not finished as soon as possible. Besides, (c) RR needs to store the KV pairs of at most three requests (CDE), while (d) only needs to store the KV pairs of at most two requests (CD or CE). The performance can be worse for more bursty requests. Second, MuxServe assigns a quota of computation resources to each shared LLM service. However, as the request traffic is dynamic, the preassigned quota increases the latency for bursty services and wastes GPUs for services at the traffic trough. Although MuxServe improves the system throughput through sharing LLM services, it neglects the latency which is usually the primary goal of LLM serving systems.

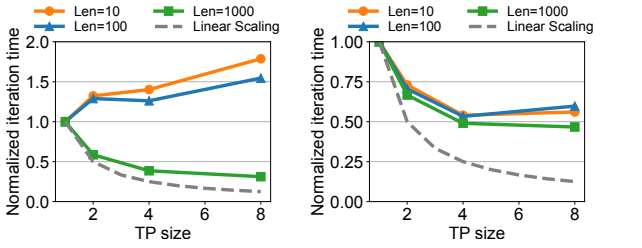
2.3 Challenges

As we analyzed in §2.2, existing sharing methods do not perform well for LLM services. We summarize the challenges in designing an efficient and latency-optimized sharing system for LLM services as follows.

Various characteristics of LLM services. LLM services have various characteristics decided by the LLM architecture, and the distributions of the input and output lengths. For example, Table 1 shows that the chatbot service (ShareGPT) has shorter inputs and longer outputs on average, while the summarization service (LongBench) has longer inputs and shorter outputs. The optimal placement and parallelism plan can be different for different LLM services. As shown in Figure 3, a larger tensor parallelism (TP) size decreases the iteration time for long inputs served by Llama2-13B, while a larger TP size may increase the iteration time for short inputs



(a) Prefill phase, Llama2-7B. (b) Decoding phase, Llama2-7B.



(c) Prefill phase, Llama2-13B. (d) Decoding phase, Llama2-13B.

Figure 3: The iteration time of the prefill phase and the decoding phase under different TP sizes. Len represents the number of input tokens and the batch size is 8.

served by Llama2-7B. Besides, the speedups of TP are also different between the prefill phase and the decoding phase. Hence, there is a critical need to consider the characteristics of LLM services for service placement and request scheduling.

Large search space of service placement. For a cluster with multiple LLM services, allocating resources properly for each service is crucial for the overall cluster performance. However, the search space of service placement is extremely large. Even without sharing and parallelism, n LLM services in a m-GPU cluster can have $\binom{m-1}{n-1}$ kinds of possible placements, which is factorial to n and m. For example, there are more than 3×10^8 placements for 16 services in a 32-GPU cluster. Parallelism configuration influences the serving speed as shown in Figure 3, and thus, we need to choose the proper parallelism configuration for each LLM service. Additionally, LLM services are different in request traffic and the sharing group can affect the performance of each LLM service in it. The parallelism configuration and the sharing group make the search space larger and more complex. Therefore, how to place multiple LLM services efficiently is challenging.

Compound requirements of scheduling. Efficiency and performance are two main requirements of the scheduling algorithm. First, the scheduling algorithm should have low time complexity, i.e., efficiency. We cannot design complex scheduling algorithms for LLM services as the inference process of requests only takes seconds or milliseconds.

Second, the scheduling algorithm should generate an execution sequence of requests with optimized serving performance. Scheduling algorithms can influence the overall performance of all services in a cluster. Note that the three scheduling algorithms in Figure 2(b)-(d) have different performances. The main reason is that the characteristics of the services are different, where the requests of S_0 are more bursty and shorter than those of S_1. Latency is one of the most important metrics for LLM services, because the output of LLM is long and users care about the response speed. To better evaluate the overall performance, we compare the average latency of request $A-E$. However, since the output lengths vary greatly, it is unfair to simply calculate the average latency of different requests. Therefore, we use normalized latency, L_N, as a metric of overall performance, which can be formulated as,

$$L_N = \sum_{s \in S} \sum_{r \in R_s} L_r / \hat{L}_s, \tag{1}$$

where S is the set of LLM services, R_s is the requests of service s, L_r is the real latency of request r, and $\hat{L}_s$ is the average request execution time of service s. The request latency L_r can be decomposed into token-level metrics, including TTFT and TPOT. Thus optimizing L_N can also improve these token-level metrics to some extent.

As we analyzed in §2.2, the commonly used scheduling algorithms for serving workloads, i.e., FCFS and RR, are not suitable for shared LLM services. As shown in Figure 2, the normalized latency L_N of (d) is $2\times$ faster than that of FCFS and $1.3\times$ faster than that of RR. Therefore, finding the latency-optimized execution order for the shared LLM services is both important and challenging.

Large and complex memory requirements. LLMs require large GPU memory to store the KV cache, the model weights, and the intermediate data, which makes it difficult to serve multiple LLMs simultaneously on one GPU. For example, a Llama2-7B model needs 14GB to store a half-precision model. Besides, modern LLM serving systems usually leverage the KV cache, and the KV pairs for only one request with $4,096$ input tokens and $4,096$ output tokens need 4GB to store. The large memory requirements of LLM services constrain the efficacy of resource sharing. Moreover, the KV cache consists of multiple cache blocks and the block shape is decided by the LLM architecture. As a result, we cannot simply share the KV cache of different LLM services. We need to maintain the KV cache carefully for the shared LLM services.

3 SeaLLM Overview

We propose an LLM service management system, SeaLLM, which enables efficient multi-LLM sharing, automatic LLM placement and replacement, and latency-optimized request scheduling. Figure 4 shows the overall architecture of SeaLLM. SeaLLM consists of a global manager and a set of local engines on the GPU nodes.

Global manager. SeaLLM's global manager is responsible for generating placement plans for LLM services, dispatching requests, and monitoring LLM services. There are three components in the global manager, i.e., placement controller, request dispatcher, and service monitor.

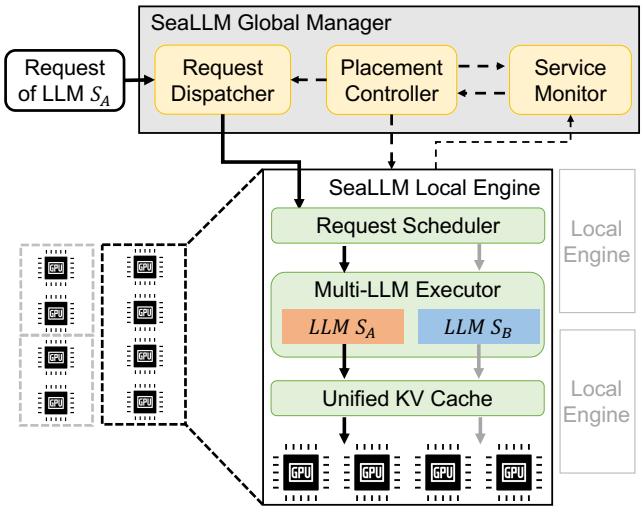


Figure 4: SeaLLM architecture.

Placement controller. The placement controller generates the placement plan for all the LLM services in the cluster. A placement plan contains the used GPUs and the parallelism configuration for each LLM service. As the request traffic of LLM services is time-varying, a fixed placement plan cannot react to the change of request traffic timely. Therefore, the placement controller generates placement plans periodically based on the historical characteristics of services. For a new LLM service, the placement controller allocates the minimum number of GPUs dedicatedly as an initial placement plan. Note that a fixed replacement interval is not optimal for the system performance and is difficult to choose manually. Specifically, a short replacement interval can lead to excessively frequent replacements, which increases system overhead. In contrast, a long replacement interval cannot react to the dynamic request traffic timely, resulting in performance deterioration. Hence, we propose an adaptive replacement algorithm to change the replacement interval automatically.

Service monitor. The service monitor monitors the LLM services in the cluster and collects their characteristics continuously. The collected data include the model information (including the iteration time of prefill and decoding phases with different TP sizes), the request information (including the average and standard variance of the input and output lengths), and the execution information (including the latency of each request and the SLO attainment of each service). These data are passed to the placement controller for periodical replacement and adjustment of the replacement interval.

Request dispatcher. When a new request arrives, the request dispatcher decides the proper local engine for the request according to its service type. If multiple local engines can serve the arrived request, we select the local engine with the least number of requests.

Local engine. The local engine is responsible for scheduling and executing the requests dispatched by the global manager. The local engines share the LLM services in time. Each local engine consists of three components, i.e., request scheduler, multi-LLM executor, and unified KV cache.

Request scheduler. The request scheduler makes scheduling decisions for the dispatched requests. There are two basic requirements for the scheduling algorithm. First, the scheduling algorithm should be fast, considering the short inference time of LLM requests. Second, the policy should provide high serving performance. Due to the unique characteristics of LLM services, simply applying FCFS or RR cannot reach optimal performance as shown in Figure 2. We propose a DB scheduling algorithm by utilizing the service characteristics to achieve better scheduling performance.

Multi-LLM executor. Each batch of requests selected by the request scheduler is sent to the multi-LLM executor for execution. The executor holds the shared models and invokes the corresponding model for inference.

Unified KV cache. In practice, the block shape of the KV cache is set according to the model architecture, e.g., the number of attention heads and layers, and the hidden size. As a result, the block shape is not identical for LLMs. SeaLLM exploits a unified KV cache to address this problem and the unified KV cache performs similarly to the KV cache for a single model.

4 Design

The goal of SeaLLM is to provide latency-optimized resource sharing for multiple LLM services in a GPU cluster. Specifically, SeaLLM needs to decide how to place the LLM services in a given cluster and how to schedule the requests of each LLM service to improve the normalized latency.

First, we formulate the problem and our goal. Assume there is a cluster with GPUs G and multiple LLM services S. For each service $s \in S$, its requests are represented as R_s and its SLO is SLO_s. We use *sharing group* to represent a set of LLM services that share the same set of GPUs. The placement of the i-th sharing group can be represented by a tuple (S_i, G_i, p_i), where S_i is the LLM services in this sharing group, G_i is the GPUs used by this sharing group, and p_i is the parallelism configuration for this group. A placement plan P for the cluster and all services S can be defined as a set of placements for sharing groups, i.e., $P = \{(S_i, G_i, p_i)\}$. With the above definition, our goal is to find the optimal placement plan P^* and the execution order of requests E^* to minimize the normalized latency under SLO constraints. We formulate the problem as follows,

$$P^*, E^* = \arg\min_{P,E} L_N(S, P, E), \tag{2}$$

where $L_N(\cdot,\cdot,\cdot)$ represents the normalized latency for services S using placement plan P and execution order E. L_N is defined in Equation 1. Besides, we have three constraints,

$$\cup S_i = S, \tag{3}$$

$$\cup G_i \subseteq G, \tag{4}$$

$$\sum_{r \in R_s} \mathbb{I}(L_r < SLO_s)/|R_s| \geq \Delta, \forall s \in S, \tag{5}$$

where $\mathbb{I}$ is an indicator function and it is one when the following condition is satisfied. Equation 3 constraints that all LLM services should be placed in the cluster. Equation 4 constraints that the number of used GPUs should not exceed the number of all GPUs in the cluster. Equation 5 denotes that the SLO attainment of all LLM services in the cluster should be larger than a threshold Δ.

The number of all placement and execution orders is factorial to the number of GPUs and services. Finding the optimal placement plan P^* and execution order E^* is a complex combinatorial optimization problem and it is almost impossible to find within an acceptable time. Hence, we split the above problem into two stages, i.e., (1) generate the placement plan for each LLM service (§4.1), and (2) given the placement plan, decide the execution order of the requests submitted to each LLM service (§4.2). Additionally, we propose an adaptive replacement algorithm for the dynamic traffic (§4.3). In the last, we introduce how we manage the unified KV cache for efficient service sharing (§4.4).

4.1 Service Placement

A placement plan includes the sharing groups, the used GPUs for each sharing group, and the parallel configurations for each sharing group. The number of all possible placements is factorial to the number of GPUs and services. For example, a cluster with 32 GPUs and 16 services has more than 10^8 possible placements even without considering the sharing groups and the parallelism configurations. Thus, it is impossible to search all placements and find the optimal one within a reasonable time. We propose a two-stage placement algorithm similar to AlpaServe [32], but we extend it for LLM services. First, we propose two heuristics to improve the searching speed because the cluster size and the number of parallel configurations of LLMs are getting larger than previous models. Second, we utilize the normalized latency and our proposed latency-optimized scheduling algorithm to evaluate the searched placement plans. Algorithm 1 in Appendix A.1 is the pseudocode of our placement algorithm. First, we partition the whole cluster into GPU groups by enumerating the parallelism configuration. Then, we allocate LLM services to the GPU groups to form sharing groups. With a simulator estimating the serving performance based on the historical request information, we can select the optimal placement with the lowest normalized latency. The simulator uses our scheduling algorithm introduced in §4.2.

Group partition. For the first stage, we enumerate all possible parallelism configurations to split the whole cluster. Existing practice mainly uses tensor parallelism for LLM inference services [29, 51], and thus we only consider tensor parallelism here. Note that our placement algorithm can be generalized to other parallelisms by enumerating their possible sizes. Assume the cluster has N nodes and each node has m GPUs, i.e., the cluster has Nm GPUs in total. The partition space has 2^{Nm-1} partitions, which is almost impossible to enumerate for a large cluster. We introduce two heuristics to prune the search space. First, we only select the power of two as the parallelism size which is commonly used in practice [32, 56]. Besides, the tensor parallelism size cannot exceed m because tensor parallelism needs large amounts of communication among GPUs and inter-node communication is slower than intra-node communication. SeaLLM can be generalized to cross-node serving by enabling other parallelisms, like pipeline parallelism. Second, we set all groups with the same tensor parallelism size. If the tensor parallelism size is too small for some services, we will merge two small groups into a larger one until all services can be placed.

Service allocation. In the second stage, we allocate LLM services to the GPU groups partitioned by the first stage. Similar to the first stage, enumerating all possible allocations is time-consuming. Therefore, we use a heuristic to speed up the process. We allocate services to the GPU group iteratively, and we select the most unserved service for each iteration. We find that only using the normalized latency to choose the most unserved service is misleading and cannot satisfy the constraint of SLO attainment (Equation 5). Because during the allocation process, some services may not be allocated yet and their requests cannot be served, which leads to a fake small normalized latency. To address this problem, we use the number of unserved requests as the main metric and the normalized latency as a secondary metric. The unserved index UI of service s can be defined as follows,

$$UI_s = \sum_{r \in R_s} \mathbb{I}(L_r > SLO_s) + \alpha L_N^s, \tag{6}$$

where L_N^s is the normalized latency of service s. With the above definition of unserved index UI_s, SeaLLM allocates the service with the highest UI_s to the GPU group which has the lowest request rate. We set $\alpha << 1$ to ensure that the SLO attainment is more important than the normalized latency. This setting can improve the overall serving performance and avoid starvation of services with long output lengths. When allocating an LLM service to a GPU group, SeaLLM first checks if the GPU memory is enough for serving the shared LLMs with profiled memory usage.

4.2 Latency-Optimized Scheduling

Given the placement, we need to decide the optimal execution order of requests for each LLM service, i.e., scheduling the requests of each LLM service to minimize the normalized latency. There are two main requirements for the scheduling algorithm, i.e., efficiency and performance. First, as the

execution time of LLM services is only seconds or even milliseconds, the scheduling algorithm should be simple and fast, without bringing significant overhead for each request. As a result, the complex scheduling algorithms for long-running tasks [21, 55] are not suitable for LLM services.

Second, the scheduling algorithm should provide high-performance scheduling decision for shared LLM services. Existing LLM engines mainly use FCFS [29, 32] or RR [18, 50] schedul ing algorithms. FCFS serves the requests in their arrival order. It introduces almost no scheduling overhead for requests. However, it is notorious for the head-of-line blocking problem, i.e., a long request in the head of the queue can block other requests. This makes FCFS unsuitable for shared LLM services, especially when the services have highly diverse characteristics. For example, in Figure 2, the requests of service S_1 have longer execution time than the requests of service S_0. Using FCFS, the request C blocks the requests D and E, which greatly increases the latency of service S_0. Some approaches use the RR scheduling algorithm. They place the requests into different queues and execute them in a round-robin pattern. These approaches can alleviate the head-of-line blocking problem, but can be inefficient for bursty and long requests with frequent preemptions. Although FastServe [50] adopts the multi-level feed-back queue to avoid frequent preemptions, it does not consider the characteristics of each shared LLM service. Thus, its improvement is limited when sharing LLM services. Additionally, these approaches also impose a heavy burden on GPU memory, because many requests are executed in turn and the KV pairs of these requests should all be stored in GPU memory.

To address the above problems, we propose the doubling budget (DB) scheduling algorithm for shared LLM services. Our key observation is that LLM services usually have different characteristics, e.g., the distributions of the input and output lengths, as shown in Table 1. Based on this observation, we can execute short requests first to decrease the normalized latency. Specifically, we set the priority of request r as

$$O_r = T'_r \hat{L}_r, \tag{7}$$

where T'_r is the estimated remaining time of r and $\hat{L}_r$ is the profiled execution time of r on one GPU. Smaller O_r represents higher priority and requests with higher priority are served first. To avoid the head-of-line blocking problem, we use preemptive scheduling and schedule requests before each iteration. The proposed scheduling algorithm is proven to reach the optimal normalized latency under certain conditions.

Theorem 1. *The proposed scheduling algorithm can minimize the normalized latency with the following assumptions: (1) requests can be preempted at any time, and (2) all requests of the same service have identical execution time.*

The proof of Theorem 1 is in Appendix A.2. Briefly, this problem is a special case of the queuing model $B|r_j, pmtn|\sum L_j/\hat{L}_j$, where B is the batch size. We can prove this theorem by contradiction. Assumption (1) can be ignored as the execution time of one iteration is usually short, especially for the decoding phase. In practice, assumption (2) is usually not the case. However, our priority O_r is still the optimal priority to minimize the expected normalized latency.

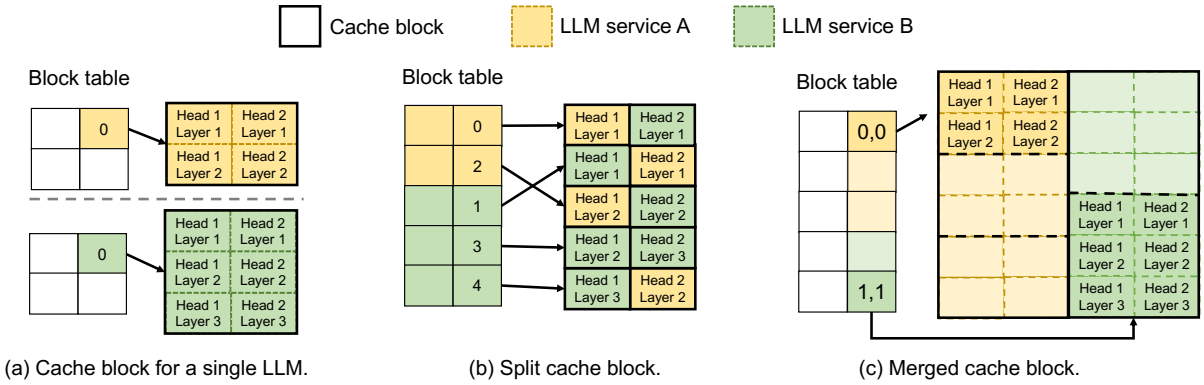
Theorem 2. *For any scheduling moment, the proposed priority O_r is optimal to minimize $\mathbb{E}(L_N)$.*

The proof of Theorem 2 is in Appendix A.2. However, in practice, some requests have extremely long outputs, leading to performance degradation. For this, we modify our DB scheduling algorithm as shown in Algorithm 2 in Appendix A.2. We introduce a budget of the execution time for each request and define the budget as $Q = (\hat{L}_s + Var_s)$, where $\hat{L}_s$ and Var_s are the average and standard variance of the execution time of requests belonging to service s. The budget Q decreases with the request execution. The priority of a request r is $O_r = Q_r \hat{L}_s$, where $r \in R_s$ and Q_r is the left budget for request r. We select the request with the highest priority, i.e., the smallest O_r, to execute first.For better GPU utilization and throughput, we batch the requests in the same phase and of the same service.If one request does not finish within its initial budget, we double its budget, i.e., $Q' = 2(\hat{L}_s + Var_s)$, and reset its priority to $Q'\hat{L}_s$.We repeat to double the budget once the budget is used up until the request is finished. We set a starvation threshold for each service and serve the starved services first. Although our scheduling policy is probabilistically optimal, it may not reach the minimum normalized latency. Besides, it is difficult to tell how far our scheduling decision is from the optimal scheduling decision, considering the unpredictable request lengths in practice.

4.3 Adaptive Replacement

The request traffic of LLM services is time-varying as shown in Figure 1. A fixed placement plan cannot respond to the changing request traffic promptly, leading to performance deterioration when bursty requests come in. To address this problem, we replace the LLM services in the cluster periodically. Periodical replacement is not new for cluster management. However, existing methods [52, 55] usually adopt fixed replacement intervals, which are difficult to determine manually. Specifically, a short replacement interval increases the system overhead of migrating services, while a long replacement interval cannot provide new placement plans timely.

SeaLLM introduces an adaptive replacement algorithm to select the proper replacement interval automatically. Our main idea is to change the replacement interval based on the difference between the achieved performance metrics and the estimated performance metrics. SeaLLM has a replacement interval I_t for the t-th replacement interval, where I_0 is an initial hyper-parameter selected by the cluster manager. Specifically, before each interval, our simulator calculates the estimated performance metrics, including the SLO attainment and the normalized performance, based on the historical data



(a) Cache block for a single LLM. (b) Split cache block. (c) Merged cache block.

Figure 5: Comparison among (a) single LLM's cache block, (b) split cache block, and (c) merged cache block.

of the last replacement interval. During the execution, the service monitor records the performance metrics for each LLM service. At the end of interval t, SeaLLM compares the real performance M_{real} and the estimated performance M_{est}, and change the replacement interval I_{t+1} as follows,

$$I_{t+1} = \begin{cases} (1-\beta)I_t & |M_{real} - M_{est}|/M_{est} > \beta \\ (1+\beta)I_t & |M_{real} - M_{est}|/M_{est} < \beta, \end{cases} \quad (8)$$

where β is a parameter in $(0,1)$. Intuitively, a larger difference between M_{real} and M_{est} indicates the current request rates vary from historical ones, suggesting that we should shorten the replacement interval. On the other hand, a smaller difference represents that the request rates remain relatively stable, allowing us to increase the replacement interval.

4.4 Unified KV Cache

Sharing multiple LLMs requires an efficient and flexible mechanism to manage the resources, especially the GPU memory. GPU memory is mainly used to store three parts, i.e., the model weight, activation, and KV cache. The model weight and the activation are relatively stable given the model information and the batch size. We allocate sufficient space for both the model weights and the peak activation usage. However, managing the KV cache for shared services is very challenging. First, the KV cache is dynamic during model execution, and separating the KV cache for each LLM can cause memory fragmentation or memory scarcity. Besides, the KV cache consists of multiple blocks whose shape is decided by the LLM architecture, including the number of hidden layers, the number of heads, and the hidden sizes. Different LLM architectures result in different block shapes. Therefore, we cannot directly share the KV cache of the shared services.

To efficiently manage the KV cache, SeaLLM introduces a unified KV cache mechanism, which changes the cache block shape to fit the shared LLMs. The hidden size is usually identical for widely-used models [18], e.g., 128 for Llamas [44] and OPT over 2.7B [54]. For other dimensions, there are mainly two possible methods, i.e., split a larger block into smaller blocks (called split method) or merge smaller blocks into a larger block (called merged method). Figure 5 shows an example of these two methods. SeaLLM adopts the merged method mainly due to three reasons. First, the merged method needs much less space for the block table than the split method. For example, the split method needs an $1,024\times$ larger block table for Llama2-7B which has 32 hidden layers and 32 attention heads, and this number increases for larger LLMs. On the other hand, the merged method only needs to add a second index for each original block to indicate the place in a merged block. Second, the merged method requires fewer read/write operations to fetch/store the same amount of KV pairs. Consequently, it achieves better locality in accessing the physical space compared to the split method, leading to improved read/write speeds. Third, although the merged method may lead to more memory fragmentations than the split method, the batched execution can alleviate this fragmentation issue.

5 Implementation

SeaLLM is implemented with approximately 12,000 lines of code in Python, C++, and CUDA, and reuses some components of vLLM [29]. There are three roles in our system, i.e., the global manager, node worker, and local engine.

Global manager. There is only one global manager for a cluster and the global manager is mainly implemented in Python. It contains a component for receiving requests, the placement algorithm, the adaptive replacement algorithm, the request dispatcher, and the service monitor. We run the placement algorithm in parallel as there are no dependencies for allocating models and evaluating the performance among different group patterns. The global manager uses gRPC [2] to communicate with other roles in the cluster. We use the Ray cluster for the distributed local engines.

Node worker. There is one node worker for each node in the cluster and the node worker is mainly implemented in Python. The node worker plays as a middleware to bridge the global manager and the local engine. Besides, the node worker monitors the states of the local engines on the same node. When abnormal states happen, e.g., one service crashes and

Baseline	Real testbed	Simulator	Difference
vLLM	13.03	13.05	0.2%
AlpaServe	11.20	11.18	0.2%
MuxServe	19.55	19.39	0.8%
SeaLLM	1.65	1.60	3%

Table 2: Comparison of the normalized latency from real testbed and simulator.

affects the shared services, the node worker will report to the global manager and help to restart the local engine. The node worker collects service characteristics from local engines, and reports to the service monitor of the global manager. When replacement happens, the node worker starts to pull model description files and weights from remote storage. The pulling process is asynchronous to minimize the effect of running services. Once all nodes in the cluster are ready for the new placement plan, the node worker migrates ongoing requests after the ongoing iteration and stops old local engines.

Local engine. Each group of shared LLM services has one local engine on the specific node. The request scheduler is implemented in Python and uses the PriorityQueue whose complexities of adding a new element and getting the smallest element are both $O(\log n)$, where n is the number of elements in the PriorityQueue. The multi-LLM executor is based on the LLM engine of vLLM and is implemented in Python, C++, and CUDA. We add support for sharing LLMs, including modifying the LLM engine and holding multiple model runners. The weights of the shared LLMs are all stored in the GPU memory for fast switch and execution. For the unified KV cache, we implement a unified block manager in Python and C++. The unified block manager can profile LLMs, calculate proper block shape, and manage the cache blocks for the shared LLM services. We use Ray workers for parallel execution and use NCCL for communication. The local engine also supports profiling and collecting the service characteristics, which are sent to the node worker and the global manager.

6 Evaluation

We evaluate SeaLLM with real-world request traces and LLM services. The evaluation shows that SeaLLM consistently outperforms the current SOTA systems. Under different request rates, SeaLLM can improve the normalized latency by up to 13.60×, the tail latency by up to 18.69×, and the SLO attainment by up to 3.64× (§6.2). Moreover, we show the efficacy of the design choices of SeaLLM (§6.3) and we also analyze the overhead of SeaLLM (§6.4).

6.1 Experiment Setup

Testbed. We conduct the testbed experiments on a cluster with 4 nodes and 32 GPUs. Each node has 8 NVIDIA A800 80GB GPUs, 128 CPUs, 2048GB of host memory, four 200Gbps InfiniBand NICs, and 400GBps NVLink bandwidth between two GPUs. We use PyTorch 2.1.2, CUDA 12.2, HuggingFace tokenizers 0.19.1, and Ray 2.35.0 for testbed experiments. We set α in Equation 6 to 0.0001 and β in Equation 8 to 0.1. In our evaluation, we share at most two LLM services per local engine because of the GPU memory limitation.

Simulator. Similar to prior work [32,55], we build a simulator to conduct the ablation study on a broader set of configurations. We profile the prefill phase and the decoding phase for each LLM service and request under different TP sizes. We also profile the memory usage of LLMs. Table 2 shows that the differences between our simulator and real testbed are less than 3%, showing the high fidelity of our simulator.

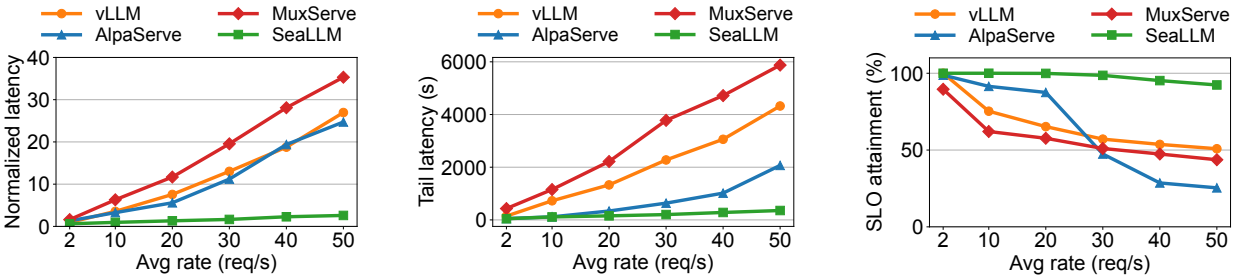
LLM service setup. Each LLM service is identified by the used LLM and the characteristics of requests. We select four commonly used LLMs with model sizes ranging from 6.7B to 70B. The selected LLMs are Llama2-7B&13B&70B [44] and OPT-6.7B [54]. The input lengths and output lengths of requests are sampled from the following real-world datasets: (1) *ShareGPT* dataset [5] is a collection of conversations between users and ChatGPT. (2) *LongBench* dataset [12] is a collection of summarization tasks. Requests in the ShareGPT dataset have longer output lengths, while requests in the LongBench dataset have longer input lengths. We use the K-means clustering algorithm to partition the requests according to their input and output lengths. For each request group, we randomly assign an LLM to form the LLM service.

Traces. Similar to prior work [32, 51], we use the Microsoft Azure function trace (MAF) [39] for arrival patterns. We round-robin functions in MAF to LLM services and generate traffic for each service. We vary the request arrival rate and select requests from a fixed duration for each trace.

Metrics. For each request rate, we measure the normalized latency (the average of end-to-end latency divided by the average execution time), the tail latency (the 99%-th latency), and the SLO attainment (the percentage of requests that finish before their SLOs). Lower normalized latency, lower tail latency, and higher SLO attainment represent better serving performance. For the SLO attainment, we set the SLO to 5× of the execution time for each request. Besides, we also evaluate two widely used metrics for token-level serving performance, i.e., the average time-to-first-token (Avg TTFT) and the average time-per-output-token (Avg TPOT).

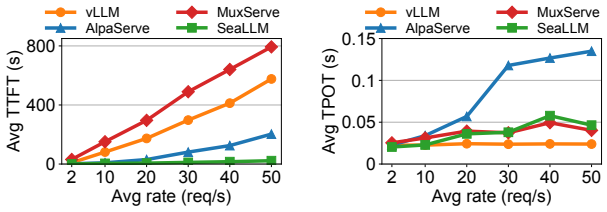
Baselines. We compare SeaLLM with the following SOTA LLM serving systems:

- *vLLM* [29] is one of the most popular LLM serving systems for a single LLM. It cannot share services and uses FCFS for scheduling. We allocate the GPUs evenly to LLM services.
- *AlpaServe* [32] proposes multiplexing for deep learning model inference. As it is not designed for LLMs with the autoregressive pattern, we implement AlpaServe with our unified KV cache and FCFS as the scheduling algorithm.
- *MuxServe* [18] is designed for sharing LLM services. It uses RR and FCFS to schedule requests. It adopts the split



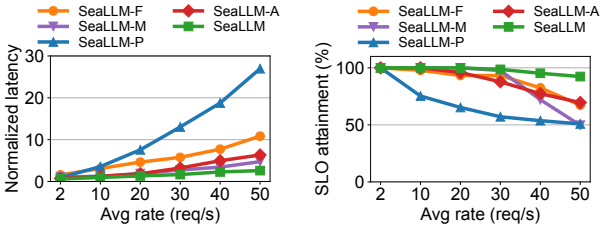
(a) Normalized latency. (b) Tail latency. (c) SLO attainment.

Figure 6: Testbed experiments on MAF traces.



(a) Avg TTFT. (b) Avg TPOT.

Figure 7: Comparison of TTFT and TPOT.



(a) Normalized latency. (b) SLO attainment.

Figure 8: Impact of the SeaLLM design choices.

method for the KV cache. We use its open-sourced code, which is also built upon vLLM, for evaluation.

6.2 End-to-End Performance

We first evaluate SeaLLM on a real cluster with 32 GPUs. Figure 6 shows the normalized latency, the tail latency, and the SLO attainment on the real-world MAF traces. We change the average arrival rate of requests from 2 to 50 requests per second. vLLM does not share LLM services and wastes GPU resources. AlpaServe uses the FCFS scheduling algorithm for the shared services, which has the head-of-line blocking problem and impairs the performance. Compared to vLLM and AlpaServe, SeaLLM reaches up to 10.38× and 9.52× lower normalized latency, 12.13× and 5.80× lower tail latency, and 1.82× and 3.64× higher SLO attainment.

MuxServe shares LLMs in both time and space dimensions, but its performance is worse than SeaLLM, mainly due to three reasons. First, MuxServe serves the services in the RR and FCFS manner, leading to lower performance when handling bursty requests. Second, MuxServe sets a fixed percentage of computational resources for each LLM service, which means that it cannot adjust to fluctuating request traffic timely, resulting in wasted computational resources. Third, MuxServe uses the split KV cache, which increases execution overhead. Compared to MuxServe, SeaLLM utilizes the service characteristics for latency-optimized scheduling, adaptive replacement for proper placement plans, and merged KV cache for lower overhead. SeaLLM improves the normalized latency by up to 13.60×, the tail latency by up to 18.69×, and the SLO attainment by up to 2.11×.

The three metrics show that SeaLLM performs better than the SOTA baselines. The normalized latency of SeaLLM is always lower than 3 and the SLO attainment of SeaLLM is always higher than 90% under different request rates, demonstrating the consistency of SeaLLM's improvement. As the service characteristics are greatly different as shown in Table 1, the lower tail latency and higher SLO attainment indicate that SeaLLM has better fairness for LLM services.

We also evaluate the Avg TTFT and the Avg TPOT, which can reflect the token-level serving performance, as shown in Figure 7. Avg TTFT is the time to get the first token and it is one of the most important metrics for online services. SeaLLM can generate the first token up to 51.98× faster than other baselines. Avg TPOT is the average time of generating each output token. Sharing LLMs inevitably increases the TPOT. SeaLLM is always faster than the temporal sharing method, AlpaServe, showing SeaLLM's efficiency. The TPOT of SeaLLM is similar to that of the temporal-spatial sharing method, MuxServe, and the dedicated method, vLLM. Note that the Avg TTFT of MuxServe is the worst mainly due to two reasons. First, its RR scheduling policy for the prefill phase increases the TTFT. Second, MuxServe restricts the used percentage of GPU computation units, which can greatly prolong the compute-bound prefill phase.

6.3 Ablation Study

Impact of SeaLLM design. To show the efficacy of our design choices, we evaluate SeaLLM with four variants: (1) SeaLLM using the FCFS scheduling algorithm (SeaLLM-F); (2) SeaLLM using the skip-join MLFQ scheduling algorithm used in FastServe [50] (SeaLLM-M); (3) SeaLLM without the placement algorithm (SeaLLM-P); and (4) SeaLLM without the adaptive replacement algorithm (SeaLLM-A); We use

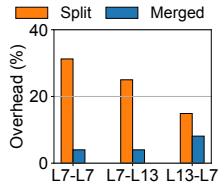


Figure 9: Impact of the unified KV cache.

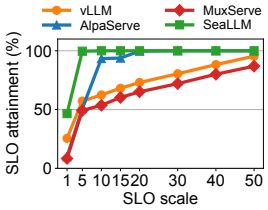
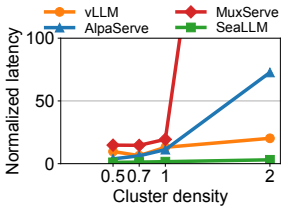
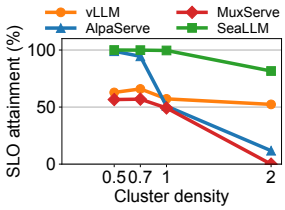


Figure 10: Impact of the SLO scale.

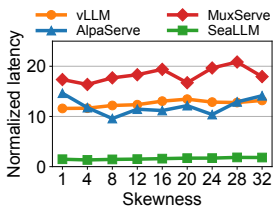


(a) Normalized latency.

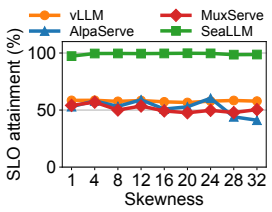


(b) SLO attainment.

Figure 11: Impact of the cluster density.

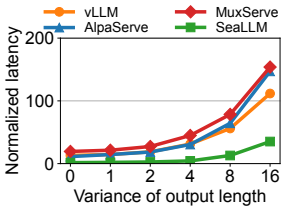


(a) Normalized latency.

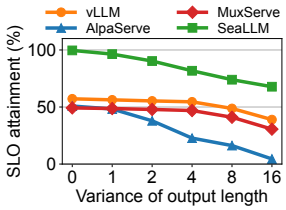


(b) SLO attainment.

Figure 12: Impact of the workload skewness.



(a) Normalized latency.



(b) SLO attainment.

Figure 13: Impact of the profiling accuracy.

the MAF traces and vary the average request rate from 2 to 50 req/s. SeaLLM-F uses the widely used FCFS scheduling algorithm, while SeaLLM-M uses one of the SOTA scheduling algorithms for single LLM services. These two variants have up to 4.17× larger normalized latency and 1.37× lower SLO attainment. Our latency-optimized scheduling algorithm outperforms FCFS because SeaLLM alleviates the head-of-line blocking problem, and outperforms skip-join MLFQ because SeaLLM integrates the service characteristics into the scheduling policy. SeaLLM-P does not share LLM services and thus is degraded to vLLM. SeaLLM-A uses the fixed placement and cannot change the placement according to the request traffic. SeaLLM-A increases the normalized latency by up to 2.44× and decreases the SLO attainment by up to 1.33×, showing the benefit of our adaptive replacement algorithm.

To study the effectiveness of SeaLLM's unified KV cache, we compare the overhead of the multi-LLM executors with the merged cache block and the split cache block as shown in Figure 9. A-B represents the overhead of running LLM A when sharing it with LLM B. L7 and L13 are short for Llama2-7B and Llama2-13B, respectively. We select three cases in which the used LLMs in L7-L7 have the same block size and the used LLMs in L7-L13 and L13-L7 have different block sizes. SeaLLM's merged cache block consistently brings lower overhead than the split cache block. The reason is that the merged cache block uses smaller block table and has better read/write locality than the split one.

Impact of the SLO scale. We also evaluate SeaLLM under different SLO scales as shown in Figure 10. The SLO for each LLM service is set to SLO scale times the average execution time of each LLM service. SeaLLM always reaches better SLO attainment than other baselines. The improvement shows the generality of SeaLLM with different SLO strictness. Besides, the lower SLO attainments of baselines indicate that they need clusters with more GPUs than SeaLLM to meet the SLO requirements. In other words, SeaLLM has better resource utilization and serving performance than baselines.

Impact of the cluster density. To investigate the impact of the cluster density, we evaluate SeaLLM over different cluster sizes with the same LLM services and request trace as shown in Figure 11. Densities $0.5-2$ represent clusters with 64 GPUs, 48 GPUs, 32 GPUs, and 16 GPUs, respectively, where a larger density number represents a more crowded cluster. Overall, SeaLLM has better performance than all the baselines, especially on the more crowded cluster. The performance of AlpaServe degrades fast with larger cluster density because its placement algorithm cannot find the optimal placement on crowded clusters. MuxServe cannot find available placement plans when the cluster density is 2.

Impact of the workload skewness. We evaluate SeaLLM with different workload skewness as shown in Figure 12. We use the number of consecutive requests from the same service as the skewness. More consecutive requests from the same service represent larger workload skewness. SeaLLM has the best performance among all baselines under different skewness. Besides, the performance for all baselines is slightly decreased when the skewness increases. This is because larger skewness means more bursty requests, which is more challenging for the serving system. The metric curves of SeaLLM do not fluctuate much, illustrating that SeaLLM is good at serving skew LLM services and bursty requests.

Impact of the profiling accuracy. As SeaLLM relies on the profiled service characteristics to make scheduling decisions, we also study the impact of the profiling accuracy. For each request, we sample the delta length from a Gaussian distribution with average 0 and standard variance from 0 to 16× of the output length. Figure 13 shows that all methods perform worse under a high variance of the output length, because

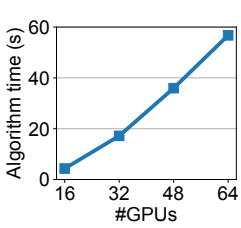


Figure 14: The running time of our placement algorithm.

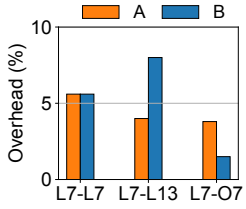


Figure 15: The overhead of the multi-LLM executor.

there are longer requests under high variance and they can impair the performance greatly. The benefits of SeaLLM get smaller under a high variance of the output length because the inaccurate profiling information affects our scheduling algorithm. However, SeaLLM still outperforms other baselines under different variances of the output length, showing the efficacy and robustness of our system design.

6.4 System Overhead

Placement algorithm running time. We evaluate the running time of our placement algorithm with different GPU numbers, as shown in Figure 14. We can speed up the placement algorithm with at most eight parallel processes, considering the possible TP sizes and the number of shared LLM services. The running time is shorter than one minute, which can be fully overlapped by running before the replacement moment.

Multi-LLM executor. Figure 15 shows the increase in iteration time when sharing multiple LLMs on the same set of GPUs. A-B represents sharing LLM service A with LLM service B. L7, L13, and O7 are short for Llama2-7B, Llama2-13B, and OPT-6.7B, respectively. We select three cases to demonstrate that the overhead is acceptable. The three cases are sharing LLMs with the same architecture (L7-L7), sharing LLMs with similar architecture but different model sizes (L7-L13), and sharing LLMs with different architectures (L7-O7). The increases in iteration time are under 4% for three out of six LLMs and under 8% for all cases.

7 Related Work

LLM serving systems. The fast development of LLMs has drawn great attention to LLM serving systems. Some systems leverage efficient operators to improve the execution speed of LLMs, e.g., TensorRT-LLM [6], LightSeq [47], Flash Attention [17], and Flash-Decoding [24]. Recent work, like FastTransformer [1], Deepspeed-Inference [11], and TGI [7], has explored intra- and inter-operator parallelism for fast serving. Orca [53] and FastServe [50] propose scheduling algorithms for LLMs. Besides, SplitWise [36], DistServe [56], and TetriInfer [25] propose disaggregation of the prefill phase and decoding phase of LLM serving. vLLM [29] manages the memory usage of LLMs. LoongServe [49], SARATHI [9], and DeepSpeed-FastGen [23] are optimized for long-context settings. These systems optimize the execution of a single LLM. Llumnix [43] notices the dynamic request traffic and reschedules requests among multiple model instances. However, Llumnix optimizes multiple instances of the same LLM service. SeaLLM explores the multiple-LLM-service setting which is orthogonal to these single-LLM optimizations.

GPU sharing. GPU sharing has been widely studied for DL models. Some existing approaches [34, 52, 55] are designed for training DL models, which usually have a repeated pattern. They are not suitable for sharing serving workloads which are time-varying and bursty. Some systems [22, 32] are designed for sharing serving workloads. These approaches do not consider the autoregressive pattern of LLM serving and directly applying them to LLM services can cause the head-of-line blocking problem. dLoRA [51] is designed for sharing LLM services. However, it is specifically designed for the multi-LoRA scenario which cannot be applied to general scenarios. MuxServe [18] is the closest work to SeaLLM, but it utilizes a scheduling algorithm combining RR and FCFS, which increases the request latency and the memory burden of GPUs. Besides, MuxServe optimizes the system throughput and preallocates GPU quota for each LLM service, which cannot react timely to the fluctuations in request traffic. SeaLLM is designed for sharing general LLM services. Besides, SeaLLM utilizes a latency-optimized scheduling algorithm to minimize normalized latency and an adaptive replacement algorithm for the changing request traffic.

Model parallelism. As the model size of LLMs is extremely large, model parallelism is required for training and serving LLMs. Tensor parallelism [38, 40] splits operators on different devices and is widely used for LLM workloads. Besides, pipeline parallelism [26, 33, 35] splits LLM layers on different devices. As the context is getting longer, sequence parallelism is introduced to LLM workloads [28, 30, 49]. The model parallelisms are usually mixed for LLM training, while for LLM serving, the most widely used parallelism pattern is tensor parallelism. Note that the placement algorithm of SeaLLM is not limited to tensor parallelism, and SeaLLM can integrate new parallelism techniques into its placement algorithm.

8 Conclusion

In this paper, we presented SeaLLM, a service-aware and latency-optimized sharing system for multiple LLMs. SeaLLM uses a latency-optimized scheduling algorithm by utilizing the service characteristics. SeaLLM exploits a placement algorithm to find the placement plan and utilizes an adaptive replacement algorithm to determine the replacement interval. To execute shared LLMs, we propose the unified KV cache to manage the memory resources. Evaluation with real-world traces and LLM services shows that SeaLLM significantly improves the normalized latency, tail latency, and SLO attainment compared to prior SOTA solutions.

References

[1] Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2019.

[2] Grpc. https://grpc.io, 2021.

[3] Multi-llms chatbot on intel developer cloud. https://github.com/intel/Multi-llms-Chatbot-CloudNative-LangChain, 2022.

[4] Multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2022.

[5] Sharegpt. https://sharegpt.com/, 2023.

[6] Tensorrt-llm. https://github.com/NVIDIA/TensorRT-LLM, 2023.

[7] Text generation inference. https://github.com/huggingface/text-generation-inference, 2023.

[8] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[9] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[10] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 2013.

[11] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *IEEE SC*, 2022.

[12] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *NIPS*, 2020.

[14] Chen Chen, Wei Wang, and Bo Li. Round-robin synchronization: Mitigating communication bottlenecks in parameter servers. In *IEEE INFOCOM*, 2019.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[16] 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, 2023.

[17] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *NIPS*, 2022.

[18] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple llm serving. In *ICML*, 2024.

[19] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[20] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *PPoPP*, 2021.

[21] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *ASPLOS*, 2023.

[22] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences. In *USENIX OSDI*, 2022.

[23] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[24] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Yuhan Dong, Yu Wang, et al. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. In *MLSys*, 2024.

[25] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NIPS*, 2019.

[27] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX NSDI*, 2019.

[28] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *MLSys*, 2023.

[29] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *ACM SOSP*, 2023.

[30] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In *ACL*, 2023.

[31] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 2022.

[32] 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 *USENIX OSDI*, 2023.

[33] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *ICML*, 2021.

[34] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient gpu memory sharing for concurrent dnn training. In *USENIX ATC*, 2021.

[35] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM SOSP*, 2019.

[36] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *IEEE ISCA*, 2024.

[37] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In *MLSys*, 2023.

[38] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC*, 2020.

[39] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX ATC*, 2020.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[41] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *ICCV*, 2023.

[42] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. *arXiv preprint arXiv:2408.00741*, 2024.

[43] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *OSDI*, 2024.

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[45] Trang H Tran, Lam M Nguyen, and Katya Scheinberg. Finding optimal policy for queueing models: New parameterization. *arXiv preprint arXiv:2206.10073*, 2022.

[46] Jan A Van Mieghem. Dynamic scheduling with convex delay costs: The generalized c| mu rule. *The Annals of Applied Probability*, 1995.

[47] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. Lightseq: A high performance inference library for transformers. *arXiv preprint arXiv:2010.13887*, 2020.

[48] 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. *arXiv preprint arXiv:2401.17644*, 2024.

[49] 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.

[50] 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.

[51] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *USENIX OSDI*, 2024.

[52] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on gpu clusters for deep learning. In *USENIX OSDI*, 2020.

[53] 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 *USENIX OSDI*, 2022.

[54] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[55] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, et al. Multi-resource interleaving for deep learning training. In *ACM SIGCOMM*, 2022.

[56] 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. In *USENIX OSDI*, 2024.

A Appendix

A.1 Placement Algorithm

Algorithm 1 shows the pseudocode of SeaLLM's placement algorithm.

Algorithm 1 Placement algorithm

Input: LLM services S; cluster GPUs G; requests R.

$P^* \leftarrow \emptyset$
$\{p'\} \leftarrow GetParallelismConfig(G,S)$
// Enumerate GPU groups
for $p \in \{p'\}$ **do**
  $P \leftarrow \emptyset$
  $B \leftarrow GenerateGPUGroups(G,p)$
  // Allocate LLM services
  while True **do**
    $UI \leftarrow SimulateUI(R,P)$
    $FindFlag \leftarrow False$
    for $b \in SortedByAvgRate(B)$ **do**
      for $s \in SortedByUI(UI,S)$ **do**
        if $b.CanAllocate(s)$ **then**
          $P.AddServices(b,s)$
          $FindFlag \leftarrow True$
          $Break$ **// To Line 17**
    if $FindFlag == False$ **then**
      $Break$
  // Simulate and get the optimal placement
  $M_{est} \leftarrow SimulatePerformance(R,P)$
  if $M_{est}.SLO > P^*.SLO$ **then**
    $P^* \leftarrow P$
  else if $M_{est}.SLO = P^*.SLO$ and
      $M_{est}.norm\_latency < P^*.norm\_latency$ **then**
    $P^* \leftarrow P$

A.2 Scheduling Algorithm

Algorithm 2 shows the pseudocode of SeaLLM's DB scheduling algorithm. The proof of Theorem 1 is as follows,

Proof. Our placement algorithm shares at most two services for each sharing group, and assume they are s_0 and s_1. The execution time on one GPU for s_0 is $\hat{L_0}$ and for s_1 is $\hat{L_1}$. Assume one execution order R' is better than the optimal execution order R^*, and the scheduling decisions are different for only two requests j and k at time t. Specifically, for time t, request j is decided to execute in R^*, but not in R'. Request k is decided to execute in R', but not in R^*. Assume the left time for j is T_j and for k is T_k. j is submitted at t_j and k is submitted at t_k. According to the sorting rule of R^*, we can get,

$$T_j\hat{L_j} < T_k\hat{L_k}. \tag{9}$$

The difference between the normalized latencies of R' and R^*

Algorithm 2 DB scheduling algorithm

Input: Requests R; LLM services S.

// Initialize
$wait\_queue \leftarrow PriorityQueue$
$decoding\_queue \leftarrow PriorityQueue$
while $True$ **do**
  // Process newly arrived requests
  while a new request r_{new} arrives **do**
    $InitBudget(r_{new})$
    $InitPriority(r_{new})$
    $wait\_queue.AddRequest(r_{new})$
  // Schedule requests
  $to\_run\_batch \leftarrow \emptyset$
  $r \leftarrow GetHighestPriorityOrStarvedRequest()$
  if $r \in wait\_queue$ **then**
    $cur\_queue \leftarrow wait\_queue$
  else
    $cur\_queue \leftarrow decoding\_queue$
  while $EnoughResource()$ **do**
    if $r.service = to\_run\_batch.service$ **then**
      $to\_run\_batch.AddRequest(r)$
      $r \leftarrow cur\_queue.Get()$
  $Execute(to\_run\_batch)$
  $UpdateBudget(to\_run\_batch)$
  $UpdatePriority(to\_run\_batch)$

is,

$$\begin{aligned} D &= \left(\frac{t+T_k-t_k}{\hat{L}_k} + \frac{t+T_k+T_j-t_j}{\hat{L}_j}\right) \\ &\quad - \left(\frac{t+T_j-t_j}{\hat{L}_j} + \frac{t+T_j+T_k-t_k}{\hat{L}_k}\right) \\ &= \frac{T_k}{\hat{L}_j} - \frac{T_j}{\hat{L}_k} \\ &< 0, \end{aligned} \tag{10}$$

where we only consider requests j and k because other requests have the same execution decisions. From Equation 10, we can get $T_j\hat{L}_j > T_k\hat{L}_k$, which is contradicted to Equation 9. If there are more than two requests having different execution orders, we can prove them pair by pair. To conclude, R^* is the optimal execution order. □

The proof of Theorem 2 is as follows,

Proof.

Lemma 1. *For a parallel single server queueing system with infinite buffer, the optimal policy is the priority policy based on the c-μ rule: The server selects the request r in queue $r^* = argmax\{c_r\mu_r\}$, where c_r is the marginal delay cost and $1/\mu_r$ is the average processing time of request r.*

Lemma 1 is proven in [45, 46]. In our scenario, the delay cost function of request i that resides τ units of time is

$$C_i(\tau) = \tau/\hat{L}_r. \tag{11}$$

The marginal delay cost function can be calculated as

$$c_i = C_i'(\tau) = 1/\hat{L}_r. \tag{12}$$

For any scheduling moment, the average processing time of request i is the estimated remaining time T_r'. Consequently, the optimal priority is

$$\begin{aligned} c_i * \mu_r &= 1/\hat{L}_r * 1/T_r' \\ &= 1/(T_r'\hat{L}_r). \end{aligned} \tag{13}$$

From the above equation, we should serve the request with the maximum $1/(T_r'\hat{L}_r)$ first, i.e., minimum $T_r'\hat{L}_r$. To conclude, our scheduling priority $O_r = T_r'\hat{L}_r$ is the optimal priority to minimize the expected normalized latency $\mathbb{E}(L_N)$ for any scheduling moment. □