M-LoRA: Efficient Serving for Concurrent LoRA Adapters with Memory-Aware Speculative Scheduler on Single GPU

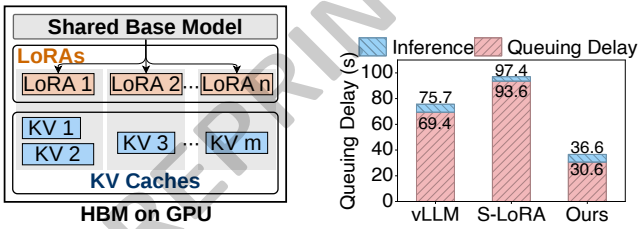
Shaolong Li[1] , **Xiang Yang**[2] , **Qi Qi**[1] , **Haifeng Sun**[1] , **Zirui Zhuang**[1] , **Bo He**[1*] , **Wanyi Ning**[3] and **Jingyu Wang**[1*]
[1]State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China
[2]Meituan, Beijing, China
[3]Tianjin University, Tianjin, China
lishaolong@bupt.edu.cn, yangxiang07@meituan.com, {qiqi8266, hfsun, zhuangzirui, hebo}@bupt.edu.cn, ningwanyi@126.com, wangjingyu@bupt.edu.cn

Abstract

Low-Rank Adaptation (LoRA) is a popular approach that enables large language models (LLMs) to quickly adapt to domain-specific tasks by adding lightweight trainable adapters. Existing multi-LoRA serving systems typically exploit parameter sharing to serve hundreds of LoRA models with a single base model. However, most systems rely on first-come-first-serve (FCFS) scheduling, which can incur severe queuing delay and lead to excessive adapter memory usage, squeezing KV cache space and reducing concurrency and throughput. To address these challenges, we propose M-LoRA, a memory-aware multi-LoRA serving system that reduces queuing delay and improves throughput through efficient request scheduling guided by fine-grained memory modeling. M-LoRA consists of three components: (1) Multi-LoRA Length Predictor, which estimates the output length and KV-cache demand of each request; (2) Memory-aware Speculative Scheduler, which dispatches requests based on predicted KV-cache and adapter memory requirements to maximize throughput under a fixed GPU memory budget; and (3) Recovery Mechanism, which corrects mispredictions while preserving generation quality. Compared to state-of-the-art LoRA serving systems, M-LoRA reduces average per-token latency by 87% and improves throughput by up to 1.7×.

1 Introduction

Large language models (LLMs) have demonstrated strong capabilities in text generation tasks. Under the pretraining-and-fine-tuning paradigm, LLMs can be efficiently adapted to domain-specific tasks. LoRA (Low-Rank Adaptation) is a widely used parameter-efficient fine-tuning method. It freezes the base model parameters and introduces small additional trainable low-rank matrices (called LoRA adapters) into each transformer layer. This design allows multiple LoRA adapters to be derived from a single base model. As the number and scale of LoRA models continue to grow, it becomes crucial for cloud platforms to serve them efficiently, which directly impacts user experience and economic profits.



(a) Parameter sharing enables a single server to serve multiple LoRA models concurrently.

(b) Queuing delay when serving 20 LoRA models with different methods.

Figure 1: Existing multi-LoRA serving methods and their bottleneck.

Recent works [Sheng *et al.*, 2023; Chen *et al.*, 2024b] have explored online multi-LoRA serving. The key to improving deployment efficiency is parameter sharing (Figure 1a). Multiple LoRA models share one copy of the base-model weights in GPU memory, while their LoRA adapters are deployed separately. Adapters are initially stored in CPU memory and loaded onto the GPU on demand. This design enables a single GPU to host hundreds of LoRA models.

Existing systems propose optimizations in memory management [Zhang *et al.*, 2025], request migration [Wu *et al.*, 2024], and heterogeneous inference [Li *et al.*, 2025a]. However, these systems use simple request-scheduling strategies. For example, vLLM [Kwon *et al.*, 2023] and Punica [Chen *et al.*, 2024b] use a first-come-first-serve (FCFS) strategy, while S-LoRA [Sheng *et al.*, 2023] employs an adapter-first strategy that prioritizes requests whose target adapter is already resident on the GPU. These strategies do not account for request length, leading to severe head-of-line blocking. As shown in Figure 1b, queuing delay can exceed 69s and account for over 90% of end-to-end latency. In such a scenario, it is well established that Shortest Job First (SJF) and Shortest Remaining Time First (SRTF) minimize the average waiting time.

*Corresponding authors.

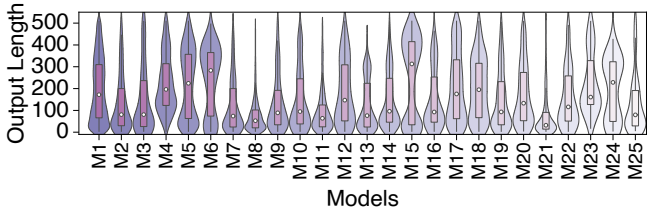


Figure 2: Output length distributions of 25 LLMs in LMSYS-Chat-1M dataset. For readability, the x-axis uses identifiers M1–M25 to denote the 25 models.

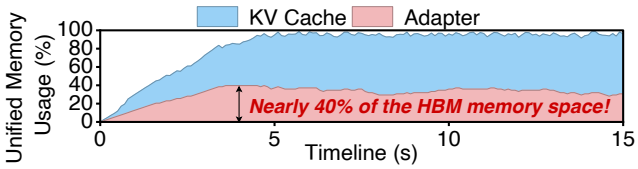


Figure 3: Memory usage of adapter and KV cache with Unified Memory management under FCFS scheduling strategy.

However, in LLM inference, the output length is unknown at scheduling time, making such length-aware policies difficult to apply directly to multi-LoRA serving.

Recent studies [Jin *et al.*, 2023; Qiu *et al.*, 2024; Fu *et al.*, 2024] have explored output-length prediction for online scheduling. Before inference, they use a lightweight predictor to estimate the generation length from the input prompt and schedule requests accordingly, approximating SJF and substantially reducing mean completion time. However, applying such length-aware scheduling to multi-LoRA serving faces two key challenges:

Challenge I: A single predictor cannot capture the output-length behaviors of diverse LoRA models. Prior work typically uses a single predictor to estimate the output length for a single LLM, whereas multi-LoRA serving deploys many LoRA models simultaneously. Even when LoRA models are fine-tuned from the same pretrained base model, their output-length distributions can differ substantially. We analyze the output-length distributions of 25 models in the LMSYS-Chat-1M dataset (Figure 2) and observe apparent distribution shifts. For example, claude-instant-1 (M6) tends to generate longer responses (peaking around 410 tokens), whereas oasst-pythia-12b (M7) is more concentrated in shorter ranges (mostly 10–200 tokens). As a result, a single predictor trained across all models cannot reliably capture the output-length distributions across them.

Challenge II: KV cache growth and adapter residency jointly create GPU memory contention. To reduce computation in autoregressive decoding, LLM systems cache intermediate attention key-value states in GPU memory as the KV cache. As generation proceeds, the KV cache grows and gradually consumes more GPU memory. A larger KV cache budget enables the system to serve more concurrent requests. Meanwhile, in multi-LoRA serving, adapters are loaded on demand and remain resident in GPU memory during execution, with their footprint varying as requests are scheduled and completed. Under a fixed GPU memory budget, KV caches and LoRA adapters directly compete. An inappropriate scheduling policy can result in LoRA adapters occupying excessive GPU memory, shrinking KV-cache space and reducing system concurrency and throughput. When using an FCFS scheduler, we observe that LoRA adapters can occupy nearly 40% of the cache space (Figure 3). Therefore, an efficient scheduling policy must jointly account for request length and LoRA adapters' memory usage.

To this end, we propose M-LoRA, a memory-aware multi-LoRA serving system that reduces queuing delay and improves throughput through efficient request scheduling guided by fine-grained memory modeling. M-LoRA consists of three components. First, the Multi-LoRA Length Predictor (MLLP) predicts request output lengths for multiple LoRA models, using a shared backbone with model-specific LoRA modules to capture model-specific length patterns. Second, we design MeSS to derive a scheduling policy that balances KV cache space and LoRA adapters' memory usage, maximizing system concurrency and throughput. Finally, M-LoRA provides an automatic recovery mechanism that handles mispredictions while preserving generation quality.

We evaluated LoRA models adapted from LLaMA [Touvron *et al.*, 2023b] models on 4090 and A100 GPUs. Evaluation results show that M-LoRA can significantly improve throughput and queuing delay with only modest overhead. Compared to the state-of-the-art multi-LoRA serving systems, M-LoRA reduces average per-token latency by 87% and improves throughput by up to 1.7×.

In summary, our contributions are as follows:

- We identify a coupled scheduling problem in multi-LoRA serving. Beyond the unknown output length of each request, request scheduling must also address GPU memory contention between growing KV caches and on-demand LoRA adapters under a fixed memory budget.
- We design M-LoRA, a memory-aware multi-LoRA serving system. Through model-specific length prediction and fine-grained memory modeling, M-LoRA derives scheduling decisions that mitigate contention between KV caches and on-demand LoRA adapters, enabling more effective GPU memory allocation under a fixed memory budget.
- We implement and evaluate M-LoRA on 4090 and A100 GPUs. Evaluation results show that M-LoRA outperforms state-of-the-art LoRA serving systems, reducing latency per token by 87% and improving throughput by up to 1.7×.

2 Related Works

Multi-LoRA serving systems. Advanced multi-LoRA serving systems adopt parameter sharing, avoiding duplicate base-model weights and reducing GPU memory demand. Punica [Chen *et al.*, 2024b] proposes the BGMV kernel to enable batched inference with rank-heterogeneous LoRA adapters. S-LoRA [Sheng *et al.*, 2023] introduces Unified Memory to manage the KV cache and LoRA adapters jointly, reducing memory fragmentation. dLoRA [Wu *et al.*, 2024] targets large-scale multi-LoRA serving in clusters, supporting dynamic adapter merging and cross-node request migration. FastLibra [Zhang *et al.*, 2025] models the dependencies between LoRA adapters and the KV cache using a tree

structure, and builds a cost model to guide efficient memory swap-in and swap-out. Toppings [Li *et al.*, 2025a] uses CPU-assisted prefill to hide adapter loading overhead and applies rank-aware scheduling for cross-node dispatch.

LLM Inference Acceleration. Advanced systems optimize LLM inference through batching [Chen *et al.*, 2024a; Yu *et al.*, 2022], memory optimization [Kwon *et al.*, 2023], GPU kernel optimization [Shah *et al.*, 2024; Chen *et al.*, 2018], and parallelization strategies [Ye *et al.*, 2024; Li *et al.*, 2023]. Speculative decoding [Li *et al.*, 2024; Li *et al.*, 2025b; Chen *et al.*, 2025] improves LLM throughput by using a lightweight draft model to generate multiple tokens in a single forward pass. KV cache compression [He *et al.*, 2025; Su *et al.*, 2025; Wei *et al.*, 2025] further reduces GPU memory footprint via low-bit quantization or compressed representations, thereby improving concurrency and throughput.

Sequence Length Prediction. Zheng et al. [Zheng *et al.*, 2024] modify prompts to let LLMs predict their own output lengths. Other studies [Jin *et al.*, 2023; Qiu *et al.*, 2024; Fu *et al.*, 2024] use BERT-based models to estimate sequence lengths. TetriInfer [Hu *et al.*, 2024] uses OPT-125M to estimate the output length of requests. However, these approaches focus on predicting output lengths for a single LLM.

3 Methodology

3.1 Overview of M-LoRA

This section presents M-LoRA, a memory-aware multi-LoRA serving system. M-LoRA aims to reduce queuing delay and improve throughput through efficient request scheduling guided by fine-grained memory modeling. M-LoRA consists of three components:

- **Multi-LoRA Length Predictor (MLLP).** MLLP is a LoRA-based predictor that estimates the output length of each request for multiple LoRA models, enabling length-aware scheduling and more accurate memory provisioning.
- **Memory-aware Speculative Scheduler (MeSS).** MeSS performs fine-grained GPU memory modeling based on predicted lengths and LoRA adapter sizes. It formulates request scheduling as an optimization problem and solves it to derive a scheduling decision that reduces queuing delay and maximizes throughput under a fixed memory budget.
- **Recovery mechanism.** To account for inevitable mispredictions, M-LoRA employs an online recovery mechanism that identifies mispredictions and adjusts execution accordingly, preserving generation correctness and completeness.

Figure 4 illustrates the workflow of M-LoRA. In the offline phase, M-LoRA trains model-specific predictors by LoRA fine-tuning on top of a shared pretrained backbone (Figure 5a), and unifies them into MLLP with shared backbone weights and model-specific LoRA modules (Figure 5b). In the online phase, MLLP predicts the output length of each incoming request with the corresponding LoRA module and inserts it into the request pool. MeSS then selects requests from the pool using Algorithm 1 to maximize throughput under the GPU memory budget. If a request exceeds its predicted length, M-LoRA re-estimates its predicted length and reschedules the request through the recovery mechanism to ensure complete generation.

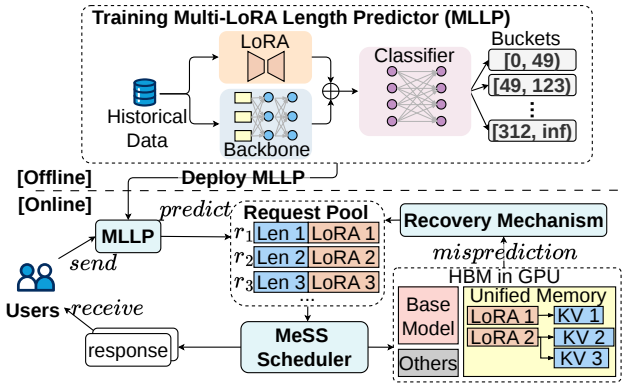


Figure 4: Overview of M-LoRA.

To jointly manage KV caches and LoRA adapters, we adopt the *Unified Memory* from S-LoRA [Sheng *et al.*, 2023]. Under Unified Memory, both KV cache and adapter weights are stored in pages: a sequence of length L consumes L pages, and an adapter with rank R consumes R pages.

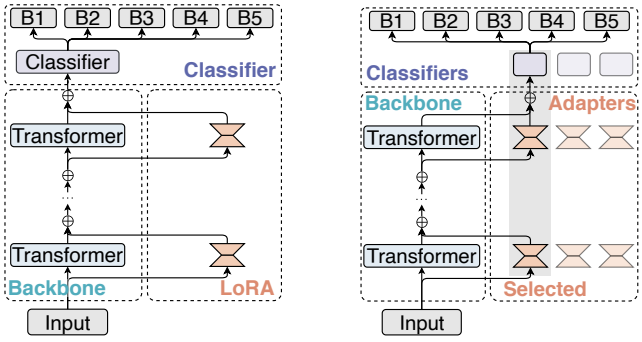
3.2 Multi-LoRA Length Predictor (MLLP)

In multi-LoRA serving, each request consumes GPU memory for both its KV cache and target LoRA adapter. Since the KV cache grows with each generated token in autoregressive decoding, the memory demand of a request is difficult to determine at arrival time. To support memory-aware scheduling, we estimate the output length of each request before execution to approximate its KV-cache demand.

We develop a Multi-LoRA Length Predictor (MLLP) to predict request output lengths in the multi-LoRA scenario. Figure 5 shows the training and inference structures of MLLP. Prior works [Jin *et al.*, 2023; Qiu *et al.*, 2024] have shown that lightweight predictors can achieve a practical accuracy-overhead trade-off for output-length estimation. Following this insight, MLLP adopts a small language model (LLaMA-3.2-1B [Touvron *et al.*, 2023b]) as its backbone. MLLP further improves accuracy and handles output length distribution shifts across LoRA models through two key designs: bucketed output length and LoRA-based fine-tuning.

Bucketed output length. As shown in Figure 2, output lengths in the LMSYS-Chat-1M dataset [Zheng *et al.*, 2023a] exhibit pronounced skew and long-tail distributions. This imbalance makes fixed-width length binning highly skewed, thereby degrading classifier training accuracy. To address this issue, we adopt a quantile-based bucketing strategy that partitions output lengths using empirical percentiles computed from the training data, resulting in more balanced buckets. In M-LoRA, we set the number of buckets to $K=5$, yielding the ranges $[0, p_{25})$, $[p_{25}, p_{50})$, $[p_{50}, p_{75})$, $[p_{75}, p_{99})$, and $[p_{99}, \infty)$.

LoRA-based fine-tuning. Beyond the within-model skewness above, output length distributions also vary widely across LoRA models. As a result, a fully shared predictor struggles to fit all target models simultaneously, leading to degraded classification accuracy. To address this issue, MLLP adopts a shared-backbone, model-specific design.



(a) Training structure offline. (b) Inference structure online.

Figure 5: The structure of the Multi-LoRA length predictor.

Specifically, we use a lightweight pretrained language model (LLaMA-3.2-1B) as the shared backbone to extract general textual representations, and train a lightweight LoRA adapter together with a classification head for each target model. Compared with fully shared predictors used in prior works [Jin *et al.*, 2023; Qiu *et al.*, 2024; Stojkovic *et al.*, 2024], our LoRA-based predictor offers two key benefits:

- **Specialization with low overhead.** Model-specific LoRA module captures per-model length characteristics while adding only a small number of parameters.
- **Scalability.** Supporting a new target model only requires training its LoRA adapter and classification head, without retraining the entire predictor.

Training and inference. MLLP consists of a training phase and an inference phase. Figure 5a shows the training structure. Since MLLP predicts length buckets, we train it as a multi-class classifier with cross-entropy loss. For each target LoRA model, we adopt a two-stage training procedure. In Stage 1, we freeze the shared backbone and optimize the corresponding LoRA adapter and classification head. In Stage 2, we freeze the LoRA adapter and fine-tune only the classification head to refine the decision boundary.

During inference (Figure 5b), we keep one shared backbone and store all model-specific LoRA modules and classification heads in GPU memory. For each request, MLLP activates only the LoRA module and heads for the requested LoRA model to predict the output-length bucket. Each model-specific adapter and classification head occupies less than 10MB, resulting in tolerable overhead.

3.3 Memory-aware Speculative Scheduler

To maximize concurrency and throughput under a fixed GPU memory budget, we propose Memory-aware Speculative Scheduler (MeSS), which formulates request scheduling as an optimization problem. We next define the problem.

Problem definition. Given a set of requests to be scheduled, each with an input length, a predicted output length, and the corresponding LoRA model and adapter size, our goal is to select the subset of requests that maximizes the batch size while satisfying GPU memory constraints.

Algorithm 1 Computing the Peak KV Cache for a Batch.

Input: Requests list: R; whether the request has been selected: X; prompt length, generated length and predicted length of requests: $L_{in}, L_{gen}, L_{pred}$;
Output: the peak pages required for the batch: P_{min}

$C \leftarrow []$
for $r \in R$ **do**
  C.append($(r, L_{pred}[r] - L_{gen}[r])$)
end for
Sort C in ascending order of the second element.
$l \leftarrow$ the length of C
$P_{min} \leftarrow 0$
for $k = 0, ..., l-1$ **do**
  $cur\_sum\_x \leftarrow 0$, $cur\_pages \leftarrow 0$
  for $i = 0, ..., k$ **do**
    $cur\_sum\_x \mathrel{+}= X[C_i]$
    $cur\_pages \mathrel{+}= (L_{in}[C_i] + L_{gen}[C_i]) \times X[C_i]$
  end for
  $cur\_pages \mathrel{+}= (L_{pred}[C_k] - L_{gen}[C_k]) \times cur\_sum\_x$
  if $cur\_pages > P_{min}$ **then**
    $P_{min} \leftarrow cur\_pages$
  end if
end for

ILP Formulation. We formulate request scheduling as an Integer Linear Programming (ILP) problem and solve it with an off-the-shelf solver. We adopt a simple but effective objective: maximizing the decoding batch size. Under a fixed GPU memory budget, a larger decoding batch typically increases concurrency and improves GPU utilization. We now describe the ILP formulation. Given the waiting requests $R_{waiting}$ and running requests in current batch $R_{running}$, with $R = R_{waiting} \cup R_{running}$, we formulate the problem as follows:

$$\max f(x) = \sum_{i \in R} x_i \tag{1}$$

subject to:

$$P_{kv} + P_{adapter} \leq P \tag{2}$$

We introduce a binary variable x_i for each request i, where $x_i = 1$ means request i is selected for scheduling; otherwise, it continues to wait. Constraint (2) represents the memory constraint, where P denotes the total number of pages of Unified Memory. Unified Memory can be decomposed into two parts: P_{kv} pages for KV caches and $P_{adapters}$ pages for adapters. The sum of P_{kv} and $P_{adapters}$ must be less than P to satisfy the memory constraints. The key challenge in modeling the memory constraint is to accurately estimate the memory required for the selected batch to complete decoding, as both the KV cache and LoRA adapters can change over time. We next describe how GPU memory is modeled.

KV cache modeling. Since we perform iteration-level scheduling, the memory allocated to shorter requests is released as soon as they finish decoding. As a result, simply summing request lengths within a batch can substantially overestimate the required KV cache space. Therefore, we design an algorithm to compute the peak KV cache memory

Queue Length	16	32	64	128	256
Execution Time (ms)	11.05	18.86	32.73	71.41	194.52

Table 1: Solver execution time under different numbers of requests.

required for the batch to complete decoding. Equation (3) formalizes this minimum requirement.

$$P_{kv} = \max_{k \in R} \sum_{i=0}^{k} (L_{in}[C_i] + L_{gen}[C_i]) \times x_{C_i} + (L_{pred}[C_k] - L_{gen}[C_k]) \times \sum_{i=0}^{k} x_{C_i} \tag{3}$$

$$x_i \in 0, 1, \forall i \in R_{waiting} \tag{4}$$

$$x_i = 1, \forall i \in R_{running} \tag{5}$$

where L_{in}, L_{gen}, and L_{pred} denote the input lengths, generated lengths, and predicted output lengths of the requests, respectively. C is the list of request indices sorted by the *remaining output length*, calculated as $L_{pred} - L_{gen}$.

Algorithm 1 implements Equation (3) by simulating KV-cache usage as shorter requests complete and release their cache. The algorithm returns the peak KV cache usage during the simulation, which corresponds to the minimum KV cache capacity required for the batch to complete the decoding phase. Lines 1–5 construct and sort the list C in ascending order of *remaining output length*. Lines 7–18 compute KV-cache usage after each request in C completes, recording the number of pages occupied by the batch at each point. P_{min} denotes the minimum number of pages required for batched inference. In addition, Constraint (5) prevents preemption by enforcing $x_i = 1$ for all currently running requests.

Adapters modeling. In batched inference with multiple LoRA models, the BGMV [Chen *et al.*, 2024b] kernel groups requests by target adapters. Thus, the parameters of each adapter need to be stored only once in GPU memory. We introduce a binary variable y_j for each adapter $j \in \{1, \ldots, M\}$, where $y_j = 1$ if adapter j is required by the selected batch and $y_j = 0$ otherwise. Let M denote the number of adapters. The adapter memory cost is modeled as follows:

$$P_{adapter} = \sum_{j=0}^{M-1} y_j \times S_j \tag{6}$$

$$\forall i \in \{0, ..., N-1\}, \forall j \in \{0, ..., M-1\}, \quad x_i * A_{ij} \leq y_j \tag{7}$$

$$\forall j \in \{0, ..., M-1\} \quad y_j \leq \sum_{i=0}^{N-1} x_i \times A_{ij} \tag{8}$$

$$\forall j \in \{0, ..., M-1\}, y_j \in \{0, 1\} \tag{9}$$

$$\forall i \in \{0, ..., N-1\}, \forall j \in \{0, ..., M-1\}, A_{ij} \in \{0, 1\} \tag{10}$$

where S lists adapter sizes in pages under Unified Memory. Let A be a matrix $\{0, 1\}^{N \times M}$ that describes the relationship between requests and adapters. $A_{ij} = 1$ if request i uses adapter j, and $A_{ij} = 0$ otherwise. Constraint (7) ensures that if request i is selected, adapter j is either already in HBM or will be loaded. Constraint (8) ensures that each adapter is stored at most once in HBM. Equation (6) can compute the total number of pages required by the selected adapters.

The complexity of the solver. We use an off-the-shelf solver [Google, 2024] to solve the ILP. Table 1 reports the solver runtime under different queue lengths. We run MeSS asynchronously on the CPU to overlap scheduling with the GPU decoding. To fully hide the scheduling latency, we restrict each solving step to the top 64 requests in the queue, which keeps the solver time below 30ms, while decoding typically takes 35-40ms per iteration. Based on this constraint, we prioritize requests in the request pool as follows:

$$p = \frac{prompt\_len + predicted\_len}{waiting\_time} \tag{11}$$

where $prompt\_len$ is the prompt length and $predicted\_len$ is the length predicted by MLLP. Intuitively, shorter requests are prioritized to admit larger batches under the memory budget, while dividing by $waiting_time$ mitigates starvation. After ranking by p, we select the top 64 requests as scheduling candidates to reduce solver time while maintaining fairness.

3.4 Recovery Mechanism

In this section, we design a recovery mechanism to handle output-length mispredictions. Mispredictions fall into two cases: (1) When the predicted length is longer than the actual output, M-LoRA over-reserves KV-cache memory, but the request can still complete decoding normally; (2) When the predicted length is shorter than the actual output, the request may be interrupted due to insufficient memory reservation, potentially degrading generation quality. Therefore, our recovery mechanism focuses on the second case to avoid affecting text generation quality.

Recovery Mechanism. During decoding, M-LoRA continuously tracks the execution state of each in-flight request. If a request reaches its predicted output length but has not generated the EOS token, M-LoRA triggers recovery: it removes the request from the current batch, re-estimates its remaining output length based on the generated tokens, and re-enqueues it into the request pool for re-scheduling by MeSS. We provide two recovery strategies: *recomputation* and *CPU-load*:

- *Recomputation.* We append the generated tokens to the input and discard the KV cache. When the request is rescheduled, we recompute the KV cache via a prefill pass, which is faster than performing multiple decoding iterations.
- *CPU-load.* We first offload the KV cache of a mispredicted request from HBM to CPU memory. When the request is rescheduled, it reallocates HBM space and copies the KV cache back to the GPU, avoiding recomputation.

Automatic recovery with a cost model. Since the two recovery strategies rely on different data paths, they exhibit different performance characteristics. *Recomputation* rebuilds KV caches using GPU compute and writes to HBM via the GPU internal bus, whereas *CPU-load* transfers KV caches between CPU memory and GPU HBM over PCIe. We evaluate both strategies by measuring their recovery overhead across different sequence lengths. We find that the overhead of both strategies grows approximately linearly with sequence length. Accordingly, we fit the measured overhead of

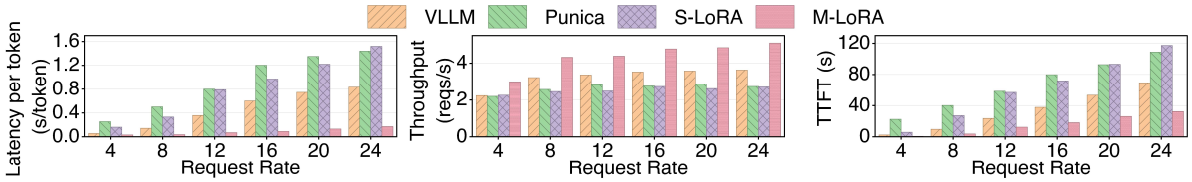


Figure 6: Latency per token, throughput, and TTFT for different systems serving LLaMA-2-7B on a 4090 GPU.

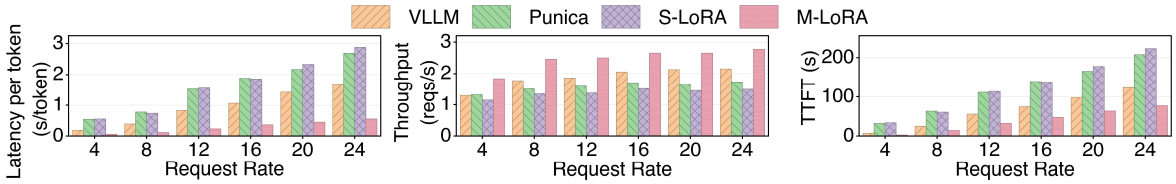


Figure 7: Latency per token, throughput, and TTFT for different systems serving LLaMA-2-13B on an A100 GPU.

Models	Size	# of Layers	Hidden Size	LoRA ranks
LLaMA-2-7B	13GB	32	4096	{8,16,32,64}
LLaMA-2-13B	26GB	40	5120	{8,16,32,64}

Table 2: Model configurations.

each strategy with a linear model. Given the sequence length of request i, we estimate the recovery overhead as follows:

$$Cost(l_i) = \min(f_{recomp}(l_i), f_{cpu\_load}(l_i)) \quad (12)$$

$$f_{recomp}(l_i) = k_{recomp} \cdot l_i + b_{recomp} \quad (13)$$

$$f_{cpu\_load}(l_i) = k_{cpu\_load} \cdot l_i + b_{cpu\_load} \quad (14)$$

where l_i represents the sequence length of request i, and f_{recomp} and f_{cpu_load} are the linear functions for the two methods, used to predict the recovery overhead based on the sequence length. We obtain the cost model by offline profiling. During online serving, M-LoRA selects the strategy with the lower estimated cost to recover the request. The fitted functions are reported in Section 4.5.

4 Evaluation

4.1 Experiment Setup

Models. As shown in Table 2, we evaluated the LLaMA [Touvron *et al.*, 2023a] series models, including LLaMA-2-7B and LLaMA-2-13B. We use several LoRA adapters for each LoRA model, with LoRA ranks ranging from 8 to 64.

Workloads. We generate workloads using LMSYS-Chat-1M dataset [Zheng *et al.*, 2023a], which contains 1M conversations from 25 real-world LLMs. To capture skewed LoRA popularity, we map LoRA adapters to these 25 models in a round-robin manner. We sample the request arrival pattern from the Chatbot Arena [Zheng *et al.*, 2023b] dataset, which records user dialogues from 20 LLMs. For each experiment, we set the request rate R and the test duration D. We sample $R \times D$ requests and linearly scale the arrival times into $(0, D]$.

Testbed. We evaluate LLaMA-2-7B on an NVIDIA RTX 4090 24GB GPU and LLaMA-2-13B on an NVIDIA A100 40GB GPU, using PyTorch 2.1 and CUDA 11.8.

Baselines. We compare M-LoRA with three state-of-the-art LoRA serving systems.

- **vLLM** [Kwon *et al.*, 2023]: vLLM uses PagedAttention to reduce memory fragmentation and statically partitions HBM for adapters and KV caches. It also supports prefix caching [Zheng *et al.*, 2023c] and chunked prefill [Agrawal *et al.*, 2023], and employs an FCFS scheduling strategy.
- **Punica** [Chen *et al.*, 2024b]: Punica is a LoRA model serving system, which implements a custom CUDA kernel to support batch inference of multiple LoRA models. Punica uses the FCFS strategy to schedule user requests.
- **S-LoRA** [Sheng *et al.*, 2023]: S-LoRA stores KV caches and LoRA adapters in *Unified Memory* and uses an adapter-first policy that prioritizes requests whose adapters are already resident in HBM.

4.2 End-to-end Performance

We first compare the end-to-end performance of M-LoRA with baselines on LLaMA-2-7B and LLaMA-2-13B models. In each experiment, we concurrently serve 20 LoRA models. The results are shown in Figure 6 and Figure 7. We use latency per token, throughput, and time-to-first-token (TTFT) as evaluation metrics. Following prior works [Yu *et al.*, 2022; Kwon *et al.*, 2023; Sheng *et al.*, 2023], we define average *latency per token* as the sum of the end-to-end latency of each request divided by the total number of output tokens.

Across all test cases, M-LoRA consistently outperforms the three baselines in latency per token, throughput, and TTFT. Specifically, it reduces latency per token by 74.6%, 87.5%, and 85.6% over vLLM, Punica, and S-LoRA, respectively. In terms of throughput, M-LoRA achieves 1.4×, 1.6×, and 1.7× improvements over the same baselines. For the TTFT, M-LoRA reduces latency by 57.4%, 81.1%, and 79.2%, respectively. These results demonstrate the effectiveness of MeSS in improving latency per token and throughput. The significant TTFT reduction further shows that M-LoRA effectively reduces queuing delay.

4.3 Predictor Evaluation

We evaluate three predictor designs for multi-model prediction on LMSYS-Chat-1M and ShareGPT [ShareGPT, 2024]

	LMSYS-Chat-1M			ShareGPT		
Predictor	Accuracy (%)	Latency (s/token)	Throughput (req/s)	Accuracy (%)	Latency (s/token)	Throughput (req/s)
BERT	47.2	0.29	3.91	45.4	0.31	3.37
MLLP+BERT	56.4	0.21	4.62	45.3	0.31	3.36
MLLP+LLaMA (Our)	**62.9**	**0.15**	**5.00**	**58.3**	**0.23**	**4.21**

Table 3: Accuracy of different predictors when serving LLaMA-2-7B, with the request rate set to 24 req/s over a 30s duration.

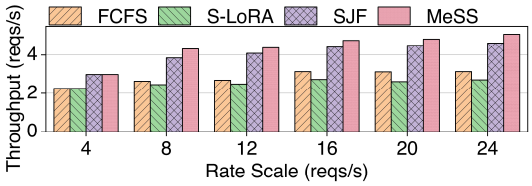
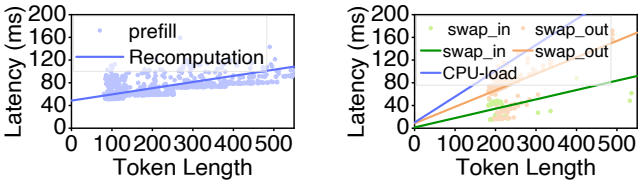


Figure 8: Throughput of different scheduling strategies on LLaMA-2-7B.



(a) Recomputation overhead. (b) CPU-load overhead.

Figure 9: Overhead of two recovery strategies when serving LLaMA-2-13B on A100. The fitted lines show the cost functions of the two strategies, which intersect at token length 113.7.

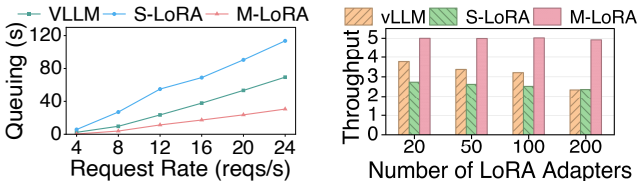
datasets: a shared BERT predictor (BERT), MLLP with a BERT backbone (MLLP+BERT), and MLLP with a LLaMA-3.2-1B backbone (MLLP+LLaMA). As shown in Table 3, MLLP+LLaMA achieves the best overall performance in terms of accuracy, latency, and throughput. These results demonstrate that the shared-backbone, model-specific design of MLLP better captures the output-length distributions of different LoRA models, leading to higher prediction accuracy and improved system performance.

4.4 Scheduling Strategies

We compare four scheduling strategies: (1) First-Come-First-Serve (FCFS). (2) The S-LoRA scheduling strategy. It prioritizes requests whose adapters are already in GPU memory. (3) Shortest Job First (SJF), which sorts requests by the predicted output length. (4) The MeSS scheduler described in Section 3.3. We evaluated the throughput of different scheduling strategies under varying request rates. As shown in Figure 8, MeSS demonstrated the best performance, improving throughput by up to $1.64\times$, $1.89\times$, and $1.12\times$ over FCFS, S-LoRA, and SJF, respectively.

4.5 Efficiency Analysis

Automatic Recovery Mechanism. We evaluate the overheads of *Recomputation* and *CPU-load* across token lengths on an A100 GPU. As shown in Figure 9, both overheads grow approximately linearly with token length. Therefore, we fit them using linear functions $f_{recomp}(x)$ and $f_{cpu\_load}(x)$. The latter includes *swap-in* and *swap-out* costs for KV-cache



(a) Reduction in queuing delay. (b) Scalability of M-LoRA

Figure 10: Efficiency of M-LoRA.

transfers between GPU and CPU. The automatic recovery mechanism selects the lower-cost strategy based on these functions and only needs reevaluation when the hardware environment changes. Among mispredicted requests, 81.4% use recomputation and 18.6% use CPU-load recovery.

Queuing delay. We compare the queuing delay of vLLM and S-LoRA. vLLM uses FCFS, while S-LoRA adopts an adapter-first policy that prioritizes requests whose target adapter is in HBM. As shown in Figure 10a, M-LoRA reduces queuing delay by 60.5% and 80.2% on average over vLLM and S-LoRA, respectively. Even with imperfect predictions, MeSS can still leverage estimated lengths to schedule requests effectively and reduce queuing delay.

Scalability. We evaluate the scalability of M-LoRA under varying numbers of LoRA adapters at 24 reqs/s. As shown in Figure 10b, S-LoRA and M-LoRA maintain stable throughput as the adapter count increases, whereas vLLM degrades significantly. This is because vLLM statically partitions memory between KV caches and LoRA adapters. As the number of adapters increases, requests spread across more adapters, increasing the number of adapters required during inference. The fixed adapter memory becomes insufficient, leading to degraded performance.

5 Conclusion

We present M-LoRA, a memory-aware multi-LoRA serving system that reduces queuing delay and improves throughput through efficient request scheduling. To handle unknown output lengths across diverse LoRA models, M-LoRA uses a model-specific length predictor to estimate the output length and KV-cache demand of each request, and applies online recovery to handle length mispredictions. To mitigate memory contention between growing KV caches and on-demand LoRA adapters, M-LoRA employs a memory-aware scheduling strategy for efficient GPU memory allocation under a fixed budget. Evaluation results show that M-LoRA reduces average per-token latency by 87% and improves throughput by up to $1.7\times$ over state-of-the-art LoRA serving systems.

Acknowledgments

This work was supported in part by the National Key R&D Program of China 2024YFE0200800, the National Natural Science Foundation of China under Grants (62471055, 62406039, 62401080, U23B2001, 62321001, 62201072, 62101064, 62171057), the Ministry of Education and China Mobile Joint Fund (MCM20200202, MCM20180101), the Fundamental Research Funds for the Central Universities (2024PTB-004).

References

[Agrawal *et al.*, 2023] 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.

[Chen *et al.*, 2018] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[Chen *et al.*, 2024a] Jinyu Chen, Wenchao Xu, Zicong Hong, Song Guo, Haozhao Wang, Jie Zhang, and Deze Zeng. Otas: An elastic transformer serving system via token adaptation. *arXiv preprint arXiv:2401.05031*, 2024.

[Chen *et al.*, 2024b] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13, 2024.

[Chen *et al.*, 2025] Fahao Chen, Peng Li, Tom H Luan, Zhou Su, and Jing Deng. Spin: Accelerating large language model inference with heterogeneous speculative models. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2025.

[Fu *et al.*, 2024] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient llm scheduling by learning to rank. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[Google, 2024] Google. Or-tools - google optimization tools. https://github.com/google/or-tools, 2024.

[He *et al.*, 2025] Ziwei He, Jian Yuan, Haoli Bai, Jingwen Leng, and Bo Jiang. Treekv: Smooth key-value cache compression with tree structures. In James Kwok, editor, *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 8086–8094. International Joint Conferences on Artificial Intelligence Organization, 8 2025. Main Track.

[Hu *et al.*, 2024] 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.

[Jin *et al.*, 2023] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. s^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

[Kwon *et al.*, 2023] 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*, pages 611–626, 2023.

[Li *et al.*, 2023] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[Li *et al.*, 2024] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees. *arXiv preprint arXiv:2406.16858*, 2024.

[Li *et al.*, 2025a] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. Toppings: Cpu-assisted, rank-aware adapter serving for llm inference. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[Li *et al.*, 2025b] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *arXiv preprint arXiv:2503.01840*, 2025.

[Qiu *et al.*, 2024] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Power-aware deep learning model serving with {μ-Serve}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 75–93, 2024.

[Shah *et al.*, 2024] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[ShareGPT, 2024] ShareGPT. Sharegpt. https://sharegpt.com/, 2024.

[Sheng *et al.*, 2023] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[Stojkovic *et al.*, 2024] 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.

[Su *et al.*, 2025] Zunhai Su, Hanyu Wei, Zhe Chen, Wang Shen, Linge Li, Huangqi Yu, and Kehong Yuan. Rotatekv: accurate and robust 2-bit kv cache quantization for llms via outlier-aware adaptive rotations. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, IJCAI '25, 2025.

[Touvron *et al.*, 2023a] 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.

[Touvron *et al.*, 2023b] 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.

[Wei *et al.*, 2025] Mingrun Wei, Yeyu Yan, and Dong Wang. Mppq: enhancing post-training quantization for llms via mixed supervision, proxy rounding, and pre-searching. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, IJCAI '25, 2025.

[Wu *et al.*, 2024] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. {dLoRA}: Dynamically orchestrating requests and adapters for {LoRA}{LLM} serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, 2024.

[Ye *et al.*, 2024] Shengyuan Ye, Jiangsu Du, Liekang Zeng, Wenzhong Ou, Xiaowen Chu, Yutong Lu, and Xu Chen. Galaxy: A resource-efficient collaborative edge ai system for in-situ transformer inference. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*, pages 1001–1010. IEEE, 2024.

[Yu *et al.*, 2022] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[Zhang *et al.*, 2025] Hang Zhang, Jiuchen Shi, Yixiao Wang, Quan Chen, Yizhou Shan, and Minyi Guo. Improving the serving performance of multi-lora large language models via efficient lora and kv cache management. *arXiv preprint arXiv:2505.03756*, 2025.

[Zheng *et al.*, 2023a] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023.

[Zheng *et al.*, 2023b] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

[Zheng *et al.*, 2023c] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. 2023.

[Zheng *et al.*, 2024] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems*, 36, 2024.