Adaptive LoRA Caching for Multi-Tenant LLM Serving

Gnana Heemmanshuu Dasari[*] *Sanjita Venkatesh Nayak*[†] *Sumit Chandrashekhar Raut*[‡]

Abstract

Production LLM serving increasingly relies on LoRA adapters to support multiple specialized models from a single base model (e.g., OpenAI's custom GPTs). However, current serving systems like Orca focus on optimizing batching and throughput for static model weights and do not address multi-adapter scenarios. Loading LoRA adapters from disk on every request introduces several milliseconds of latency, while keeping all adapters in GPU memory is too expensive when serving hundreds of specialized models. There is thus a tradeoff between latency and memory efficiency in existing systems.

We implement an adaptive caching system that effectively manages LoRA adapter lifecycle in memory based on request patterns and resource constraints. Our system implements multiple cache eviction policies (LRU, LFU) and introduces a workload-aware preloading mechanism that predicts hot adapters based on request history. The novelty lies in treating adapter management as a first-class optimization problem in LLM serving. While Orca optimizes for large model inference, we optimize for the multi-variant serving paradigm that is increasingly common in production. We measure P50/P99 latency, cache hit rates, memory utilization, and throughput under realistic multi-tenant workloads with varying adapter popularity distributions.

1 Introduction

The rapid growth of large language models (LLMs) such as GPT-3 has led to a surge in deployment scenarios that require many domain- or task-specific variants of a single base model. Full fine-tuning of each variant is increasingly impractical; parameter-efficient methods such as Low-Rank Adaptation (LoRA) mitigate training and storage costs by injecting lightweight, low-rank matrices into frozen model weights [8]. LoRA enables many compact adapters to be stored and swapped without retraining the full model, making it attractive for multi-tenant and personalized deployments.

However, the operational complexity of *serving* many LoRA adapters creates an orthogonal systems challenge. Loading adapter weights from disk on demand introduces tens to hundreds of milliseconds of extra latency per request, while keeping large numbers of adapters resident in GPU memory is infeasible at scale. Prior LLM serving systems have optimized other parts of the inference pipeline — for example, batching and iteration-level scheduling for static weights [13] or KV-cache paging [10] — but they do not directly treat adapter residency management as a primary optimization objective.

Recent systems have begun to address multi-adapter scenarios: S-LoRA presents a unified paging approach to fetch adapters from main memory to GPU memory on demand and demonstrates how unified memory pools and custom kernels permit thousands of adapters to be served [12]; Chameleon explicitly uses adapter caching combined with adapter-aware scheduling to reduce adapter load overheads in many-adapter environments [9]; and FASTLIBRA proposes dependency-aware caching of both LoRA adapters and KV caches to optimize time-to-first-token and swapping decisions [14]. These advances show that adapter management is an active area of systems research, and that caching/paging is a practical lever for improving end-to-end latency.

We present a focused exploration of **adaptive LoRA caching** as a practical, implementable approach for LLM serving in multi-adapter scenarios. Our system is intentionally simple and reproducible: it evaluates classic cache eviction policies (LRU, LFU), combined with lightweight workload prediction (exponential moving average of adapter request rates) and workload-aware preloading. The primary goal of our project is empirical: measure how much latency and throughput improvement can be achieved by these relatively lightweight techniques under realistic multi-tenant workloads (uniform, Zipfian, and bursty distributions) and constrained GPU memory budgets. While not intended to surpass fully featured systems like S-LoRA or Chameleon, our project characterizes how much benefit straightforward caching + prediction delivers and identifies practical tradeoffs for future extensions.

2 Motivation

Parameter-efficient adapters such as LoRA have lowered the barrier to creating many specialized models from a shared base, enabling use cases like tenant-specific assistants, per-customer personalization, and on-demand domain specialization. In production, however, serving these adapters creates a new bottleneck:

- **Adapter load latency.** Loading adapter weights from storage or main memory into GPU HBM on the critical path adds nontrivial latency (tens–hundreds of milliseconds), which can dominate requests for short context or single-turn tasks.

[*]ghdasari@uci.edu
[†]nayaksv@uci.edu
[‡]scraut@uci.edu

- **Memory pressure.** GPU HBM is limited; storing all adapters in GPU memory is prohibitively expensive when serving hundreds or thousands of adapters concurrently.
- **Workload heterogeneity.** Access patterns across adapters are often heavy-tailed (a small set of hot adapters receive most traffic, while many remain cold) and can be bursty, which complicates naive preloading strategies.

These constraints make adaptive memory management critical for latency-sensitive multi-tenant deployments. Systems such as Orca [13] and vLLM [10] demonstrate the value of careful scheduler and KV-cache management for throughput and latency; S-LoRA [12] and Chameleon [9] directly target the adapter residency problem using paging, caching, and adapter-aware schedulers; and FASTLIBRA [14] explicitly models dependencies between adapters and KV caches to guide swap decisions. These works suggest that caching and scheduling are effective levers, but they also leave room for simpler, engineerable approaches that can be implemented on modest infrastructure.

Therefore, a lightweight, easy-to-deploy adaptive caching layer that combines classic eviction policies (LRU/LFU) with a small workload-prediction component can be valuable as:

1. a reproducible baseline for academic evaluation and comparison with more complex systems;
2. a practical checkpoint implementation for teams that cannot integrate complex paging stacks or custom CUDA kernels; and
3. a way to quantify the returns to adding more sophisticated components (dependency models, learned predictors, adapter compression) in future work.

3 Related Work

Parameter-efficient adaptation LoRA (Low-Rank Adaptation) introduced an efficient mechanism to fine-tune large Transformer models by injecting low-rank update matrices and freezing the base model; LoRA enables storage-efficient adapters that can be swapped at runtime with negligible inference overhead in the merged-weight setting [8]. Other parameter-efficient techniques include prefix-tuning and adapter modules; the distinctions between these methods (compute/latency overhead, storage, and intrusiveness) influence serving decisions.

LLM Serving Systems Large Language Model (LLM) serving has become a critical systems problem as both model size and user concurrency have grown rapidly. Traditional serving frameworks such as **Orca** [13] and **vLLM** [10] primarily focus on optimizing batching, KV cache management, and throughput for single-model deployments. Orca introduced iteration-level scheduling and selective batching to improve throughput and latency for large Transformer models in distributed settings, while vLLM focuses on efficient attention and KV-cache reuse to reduce memory and compute inefficiencies in LLM serving. However, these systems assume static model weights and do not account for adapter-based fine-tuning paradigms that are increasingly prevalent in production. Recent systems such as **AlpaServe** [5] and **FastServe** [7] extend these ideas by exploring multi-tenant inference with dynamic resource partitioning and workload-aware scheduling. While these approaches improve GPU utilization across heterogeneous requests, they still assume all requests operate on the same base model, limiting their applicability to multi-adapter scenarios.

Multi-Adapter Serving The **S-LoRA** framework [12] demonstrated that serving thousands of adapters is feasible by using unified paging and merging strategies with custom CUDA kernels and shared GPU memory. Systems such as **LoRAX** [11] and **Punica** [2] extend this idea to support dynamic adapter multiplexing, but they assume either full in-memory residency or on-demand loading from disk, without intermediate caching layers. More recently, **Chameleon** [9] addressed many-adapter environments through adaptive caching and scheduling policies designed to reduce head-of-line blocking and improve request concurrency. Similarly, **FASTLIBRA** [14] introduced unified management for LoRA adapters and KV caches, emphasizing the importance of treating adapter swapping and inference context reuse as joint optimization problems.

Caching and Paging Strategies Keeping all adapters resident in GPU memory leads to excessive memory overhead, while loading adapters on demand incurs substantial cold-start latency—often between 100 and 500 milliseconds per request [1]. **Agullo et al. (2025)** [4] proposed an analytical framework for optimal adapter caching in multi-tenant environments, modeling memory-latency trade-offs under stochastic workload assumptions. Their results show that adaptive caching can significantly reduce latency when adapter popularity follows a Zipfian distribution, a common characteristic of production workloads. In parallel, **Wang et al. (2025)** [6] introduced *Block-Diagonal LoRA*, which reduces communication overhead in tensor-parallel adapter serving and highlights how adapter structure can influence paging performance. Industrial systems have also begun to address similar challenges. **AWS SageMaker** [1], **Ray Serve** [3], and **Predibase's LoRA Exchange** [11] discuss practical trade-offs between cold-load latency and GPU memory limits.

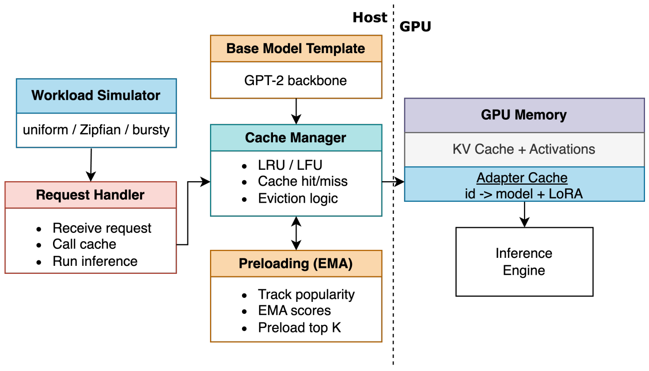

Figure 1: Architecture overview.

4 Design

Figure 1 presents an overview of our system architecture for adaptive LoRA adapter caching. The system is composed of five core components: a workload generator, a request handler, a cache manager, an optional preloading module, and the GPU-resident inference path. Together, these components emulate a lightweight multi-tenant LLM serving environment suitable for small-scale academic or research settings. We describe each component below.

1. **Workload Simulator.** The workload simulator generates sequences of inference requests following configurable distributions such as uniform, Zipfian, or bursty patterns. Each request is a tuple of the form `(adapter_id, input_text)`. These workloads emulate multi-tenant request patterns observed in practical systems such as S-LoRA and Chameleon, but scaled to a smaller adapter set.

2. **Request Handler.** The request handler receives each generated request and delegates adapter retrieval to the cache manager. Once a full model corresponding to the adapter_id is obtained (either from cache (GPU-resident) or loaded from disk (CPU-resident template)), the handler performs tokenization and executes a forward pass on the GPU. This component abstracts away caching details from the inference path, mirroring the separation of control and compute used in larger serving systems.

3. **Cache Manager.** The cache manager is the central component of the system. It supports multiple eviction strategies, including LRU and LFU. Upon receiving an adapter_id from the request handler, it checks whether the corresponding full model (base GPT-2 backbone + LoRA adapter weights) is already stored in GPU memory. If present, the model is returned immediately (cache hit). If absent (cache miss), the manager loads the LoRA adapter from disk, attaches it to the base GPT-2 model template (stored once on CPU memory), moves the resulting combined model to GPU memory, and inserts it into the cache. If the cache is full, the eviction policy determines which existing model is removed; its GPU memory is then reclaimed using standard CUDA memory management. This design isolates GPU memory pressure to only the LoRA-augmented models currently in use.

4. **Preloader (Optional).** The preloader augments the cache manager with workload-awareness. It maintains exponentially weighted popularity scores for all adapters based on recent request patterns. Periodically (e.g., every N requests), it selects the top-K most popular adapters and proactively loads them into the cache if they are not already present. This mechanism reduces cold-start latency in bursty or skewed workloads and emulates simplified forms of the predictive paging used in larger systems, but without the scheduling complexity.

5. **GPU-Resident Adapter Cache and Inference.** On the GPU side, the adapter cache stores full PEFT models that combine the GPT-2 backbone with the corresponding LoRA adapter weights. These GPU-resident models are directly consumed by the inference engine, which simply executes the model's forward pass using PyTorch operations on CUDA. The inference engine is not a separate module; rather, it represents the GPU compute path that processes the cached model and returns logits to the request handler. By storing complete adapter-specific models in GPU memory, we avoid the repeated CPU-to-GPU transfer and adapter loading overhead that dominates the no-cache baseline.

5 Evaluation

We use LoRA adapters trained on three standard NLP datasets: **IMDB** (binary sentiment classification), **AGNews** (four-class news topic classification), and **BoolQ** (yes/no question answering). For each dataset, a subset of 10,000 examples was sampled from the HuggingFace splits and preprocessed using the GPT-2 tokenizer, with maximum sequence lengths of 128 tokens (IMDB, AGNews) and 256 tokens (BoolQ). All experiments use **GPT-2** as the base model, implemented in **PyTorch** and executed on an **NVIDIA T4 GPU** for preprocessing, training, and evaluation.

Before evaluating caching policies, we measured the standalone performance of loading and running LoRA adapters without caching. We found that adapter **load time dominates the total latency**, ranging from approximately 33 ms for rank-2 adapters to 53 ms for rank-32 adapters. The inference time remains relatively constant at 12-13 ms across all datasets and ranks, confirming that the load of the adapter loading is the main bottleneck. This trend is reflected in the tail latency: P99 values reach 70-95 ms for large-rank adapters, although inference performs consistently.

These results highlight that any cache miss incurs a heavy cost, particularly for larger adapters, and therefore motivate the use of caching policies to reduce the frequency of the miss. The baseline establishes a worst-case scenario (0% hit rate), against which we evaluate LRU, LFU, and EMA-based prefetching policies in later experiments. Predictive methods such as EMA-prefetching become especially attractive because they aim to eliminate cold misses entirely by proactively loading adapters that are trending upward in popularity.

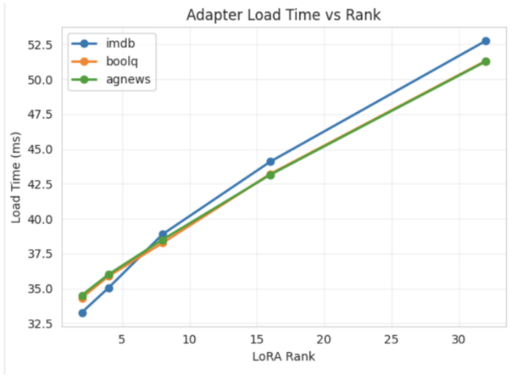

Figure 2: Adapter Load Time vs Rank

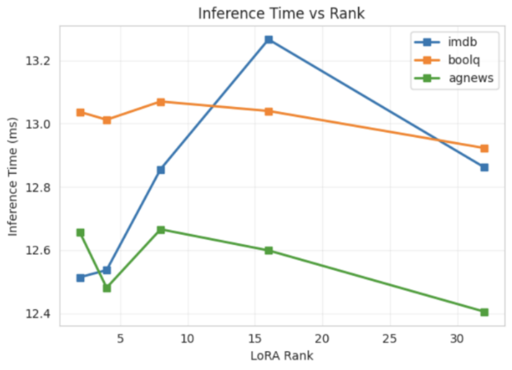

Figure 3: Inference Time vs Rank

Inference time (compute needed to run the adapter) and adapter-load time (the cost of fetching the right adapter into GPU memory) are the main components of latency. As request patterns shift, smarter cache policies reduce adapter-load overhead, making overall latency more stable.

Figure 4 illustrates the P99 latency under a bursty workload. While the inference time (faded regions) remains constant across all policies, the adapter-load time (solid regions) varies significantly. This motivates a need for a cache layer to reduce adapter load time.

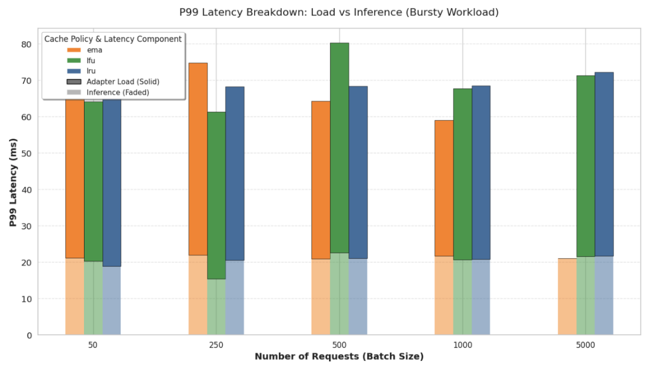

Figure 4: P99 Latency breakdown: Load vs Inference.

5.1 Cache Performance Analysis

Here, we evaluate the effectiveness of three caching strategies— **LRU**, **LFU**, and **EMA**—under different workloads. We focus on three key metrics: *hit rate*, *cache size sensitivity*, and *P99 tail latency*. The results illustrate the fundamental trade-offs between eviction policies and reveal that EMA, although highly accurate at identifying future adapter usage, suffers from increased latency due to synchronous prefetching in our implementation.

5.1.1 Cache Size Impact on Hit Rate

Figure 5 shows how hit rate improves as cache size increases from 2 to 8, evaluated across uniform, zipfian, and bursty workloads. Across all workloads and cache sizes, EMA consistently achieves the highest hit rate. This is expected: the exponential moving average gives more weight to recent access patterns, allowing EMA to track adapter popularity more accurately than the frequency-based LFU or the recency-only LRU policy.

The effect of cache size is monotonic: larger caches reduce misses across all policies. The improvement is most pronounced for policies under the zipfian workload, where heavy reuse of a small subset of adapters allows all policies to benefit from additional cache capacity.

5.1.2 Hit Rate by Workload Type

Figure 6 summarizes average hit rates for each policy under uniform, bursty, and zipfian workloads. EMA again outperforms both LFU and LRU, highlighting its ability to adapt to both frequency-driven and recency-driven access patterns.

5.1.3 Cache Hits and Hit Rate

Figure 7 depicts the cumulative cache hits and the corresponding hit rate as the number of requests increases from 0 to 1000. The EMA policy consistently demonstrates superior

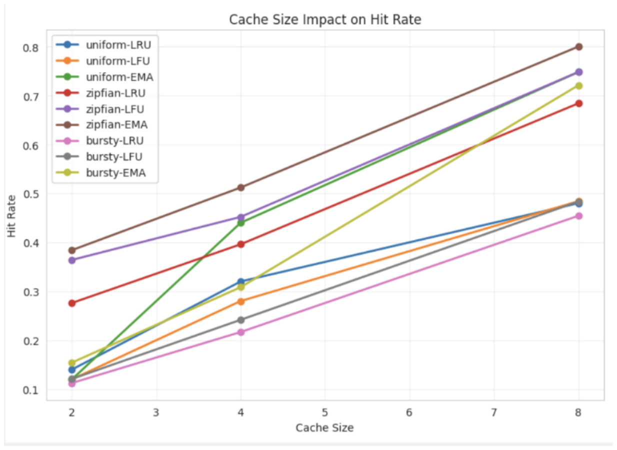

Figure 5: Effect of cache size on hit rate for LRU, LFU, and EMA under different workloads.

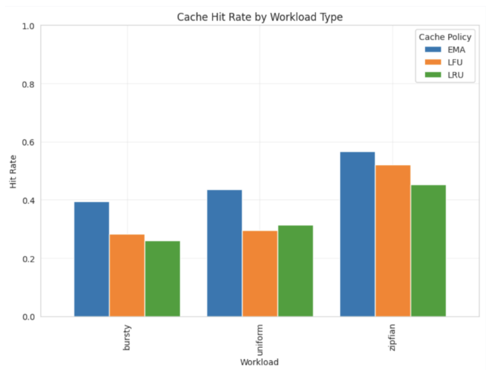

Figure 6: Hit rate across different workloads. EMA achieves the highest hit rate across all workload types.

performance compared to both LFU and LRU, exhibiting a steep linear growth in total hits.

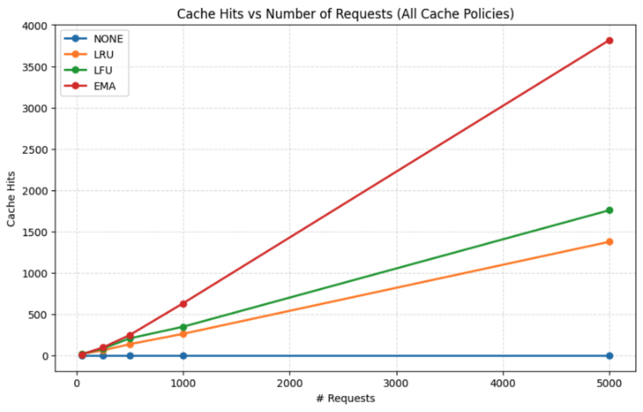

Figure 7: EMA achieves a significantly higher number of hits and sustains a higher cache hits as the request count scales.

5.1.4 P99 Latency by Cache Policy

Despite achieving the highest hit rate, EMA does *not* provide the lowest tail latency. As shown in Figure 8, EMA exhibits the **highest P99 latency** among the three policies. This result is explained by our implementation: EMA performs **synchronous prefetching**, meaning that predicted adapters are loaded from disk during the foreground request path. Since loading a LoRA adapter from disk to GPU takes 35–50 ms, synchronous prefetching adds substantial latency to the critical path, inflating P99.

In contrast, LRU and LFU do not prefetch and therefore avoid this overhead, resulting in lower P99 latency despite lower hit rates. These results highlight that prefetching improves cache accuracy but must be implemented *asynchronously* to avoid harming tail latency.

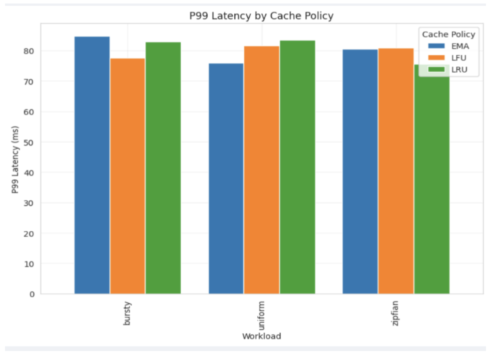

Figure 8: P99 latency for LRU, LFU, and EMA. EMA exhibits the highest tail latency due to synchronous prefetching overhead.

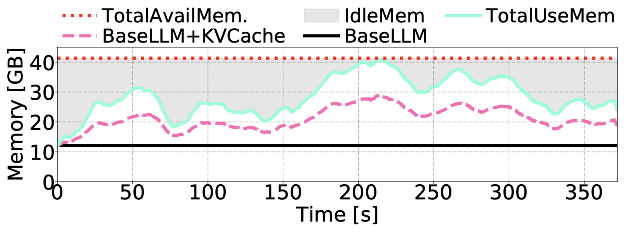

Figure 9: Memory usage over time for different parts of the computation: as displayed in the Iliakopoulou et. al. [9]

6 Discussion

Our results highlight a clear set of trade-offs across caching strategies.

- **EMA-based prefetching achieves the highest hit rate** across all workloads and cache sizes.
- **EMA also produces the worst P99 latency** due to synchronous prefetching occurring on the request path.

However, if we use Pre-warm EMA scores (prefetch policy) latency can improve.

- **LFU has a high hit rate under Zipfian workloads**, where a small set of adapters dominates.

In a real serving system, EMA prefetching would be implemented asynchronously so that preloads do not block incoming requests. Our results therefore illustrate both the potential of predictive caching and the importance of decoupling prefetching from critical inference paths.

More broadly, multi-tenant LLM serving with LoRA adapters introduces several challenges beyond caching. The original LoRA approach [8] reduces fine-tuning cost for a *single* adapter, but switching across many adapters increases latency and decreases throughput. S-LoRA [12] addresses this by batching heterogeneous requests, storing all adapters in host memory, and introducing UnifiedPaging—inspired by vLLM's PagedAttention [10]—to place active adapters in the KV cache while avoiding fragmentation. S-LoRA also employs Orca [13] for iteration-level scheduling.

Iliakopoulou et. al. [9] further observe that adapter rank variability, high-bandwidth adapter movement, and head-of-line blocking degrade performance. Even asynchronous loading increases time-to-first-token (TTFT), and decoupling the batching of the base model and adapters in S-LoRA improves throughput but increases single-query latency. Figure 9 from their work illustrates how GPU memory fluctuates under production demands. Unlike S-LoRA, Iliakopoulou et. al. empower Chameleon with a *GPU-side* cache that dynamically resizes and uses a composite eviction score (recency, frequency, adapter size). The system cooperates with a scheduler and uses admission control based on input size, output size, and adapter size. Theirs is the first work to the best of our knowledge that employs a GPU-side cache for production level workloads.

While production systems such as S-LoRA and Chameleon build sophisticated, multi-component pipelines, our simplified GPU-resident cache provides an accessible baseline for understanding these trade-offs. Several promising directions remain. For example, while Chameleon allows tuning coefficients in its eviction scoring function, an online mechanism that dynamically adjusts these coefficients based on workload patterns could further improve adaptivity.

7 Conclusion

We implemented and evaluated a lightweight GPU-side caching system for LoRA-based multi-tenant LLM serving. Through controlled experiments on uniform, Zipfian, and bursty workloads, we observed that LRU and LFU yield complementary strengths, while EMA-based prefetching substantially improves hit rate at the cost of higher tail latency. These results emphasize the importance of both workload characteristics and prefetch strategy design. Although simplified relative to production systems such as S-LoRA and Chameleon, our implementation captures the core challenges of adapter switching, eviction, and predictive caching, offering a practical foundation for researchers and students working with LoRA-based serving on limited hardware.

References

[1] AWS Machine Learning Blog. Efficient and cost-effective multi-tenant lora serving with amazon sagemaker, 2024. Available at https://aws.amazon.com/blogs/machine-learning/efficient-and-cost-effective-multi-tenant-lora-serving-with-amazon-sagemaker/.

[2] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *arXiv preprint arXiv:2310.18547*, 2023.

[3] Anyscale Docs. Serving multiple lora adapters with ray serve, 2024. Available at https://docs.anyscale.com/llm/serving/multi-lora.

[4] Agullo et al. Maximizing gpu efficiency via optimal adapter caching: An analytical approach for multi-tenant llm serving, 2025. arXiv:2508.08343.

[5] Li et al. Alpaserve: Statistical multiplexing with model parallelism for llm inference, 2023. arXiv:2311.06647.

[6] Wang et al. Block-diagonal lora for eliminating communication overhead in tensor parallel lora serving, 2025. arXiv:2510.23346.

[7] Zheng et al. Fastserve: Dynamic scheduling for efficient multi-tenant llm inference, 2024. arXiv:2401.12143.

[8] Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

[9] Nikoleta Iliakopoulou, Jovan Stojkovic, Chloe Alverti, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Chameleon: Adaptive caching and scheduling for many-adapter llm inference environments. *arXiv preprint arXiv:2411.17741*, 2024.

[10] Woosuk Kwon, Sang-Woo Lee, Jaewoong Kim, et al. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

[11] Predibase. Lora exchange: Serving hundreds of fine-tuned llms efficiently, 2024. Available at

https://predibase.com/blog/lora-exchange-lorax-serve-100s-of-fine-tuned-llms-for-the-cost-of-one.

[12] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters. *Proceedings of the Machine Learning and Systems (MLSys) Conference*, 2024.

[13] Guangyi Yu, Hyungmin Jeong, et al. Orca: A distributed serving system for transformer-based models. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[14] 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 (fastlibra). *arXiv preprint arXiv:2505.03756*, 2025.