Chameleon: Adaptive Caching and Scheduling for Many-Adapter LLM Inference Environments

Nikoleta Iliakopoulou
University of Illinois
Urbana-Champaign, IL, USA
nmi4@illinois.edu

Jovan Stojkovic
University of Illinois
Urbana-Champaign, IL, USA
jovans2@illinois.edu

Chloe Alverti
University of Illinois
Urbana-Champaign, IL, USA
xalverti@illinois.edu

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

Hubertus Franke
IBM Research
Yorktown Heights, NY, USA
frankeh@us.ibm.com

Josep Torrellas
University of Illinois
Urbana-Champaign, IL, USA
torrella@illinois.edu

Abstract

The effectiveness of LLMs has triggered an exponential rise in their deployment, imposing substantial demands on inference clusters. Such clusters often handle numerous concurrent queries for different LLM downstream tasks. To handle multi-task settings with vast LLM parameter counts, Low-Rank Adaptation (LoRA) enables task-specific fine-tuning while sharing most of the base LLM model across tasks. Hence, it supports concurrent task serving with reduced memory requirements. However, existing designs face inefficiencies: they overlook workload heterogeneity, impose high CPU-GPU link bandwidth from frequent adapter loading, and suffer from head-of-line blocking in their schedulers.

To address these challenges, we present *Chameleon*, a novel LLM serving system optimized for many-adapter environments. Chameleon introduces two new ideas: adapter caching and adapter-aware scheduling. First, Chameleon caches popular adapters in GPU memory, minimizing adapter loading times. For caching, it uses otherwise idle GPU memory, avoiding extra memory costs. Second, Chameleon uses a non-preemptive multi-queue scheduler to efficiently account for workload heterogeneity. In this way, Chameleon simultaneously prevents head of line blocking and starvation. Under high loads, Chameleon reduces the P99 and P50 TTFT latencies by 80.7% and 48.1%, respectively, over a state-of-the-art baseline, while improving the throughput by 1.5×.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning**.

Keywords

LLM inference, LLM serving systems, LoRA adapters, Adapter caching, Multi-queue scheduling

ACM Reference Format:
Nikoleta Iliakopoulou, Jovan Stojkovic, Chloe Alverti, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2025. Chameleon: Adaptive Caching and Scheduling for Many-Adapter LLM Inference Environments. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756083

1 Introduction

Generative Large Language Models (LLMs) have seen an exponential growth in recent years [2, 8, 20, 32, 41, 53, 61, 66, 68]. They have become integral to numerous technologies and applications [3, 13, 33, 42, 57]. As their popularity increases, the number of online queries received by datacenter inference clusters continuously grows [21]. These queries typically target a variety of downstream tasks, e.g., chat-bot conversation, coding, or text summarization. These different tasks require different or special-purpose fine-tuned LLMs to achieve their highest accuracy. Unfortunately, this requirement imposes a large hardware [41] and energy [51] tax on datacenters, as each of these models typically requires large memory and, thus, many GPUs, to store its many parameters.

To alleviate this problem, adapter-based techniques such as Low-Rank Adaptation (*LoRA*) [17, 58], have been explored. These methods fine-tune a small subset of a base model's parameters for every task. Recent serving systems [4, 49] leverage this technique. They decouple the base model and the fine-tuned adapter parameters, allowing different colocated LLMs to share the base model. This enables serving potentially hundreds of LoRA fine-tuned LLMs at a much lower memory cost.

However, our characterization of this environment shows that adapter-based LLM serving systems exhibit two challenges that substantially reduce performance. First, inference clusters have to orchestrate the adapters required by incoming requests as they are being scheduled. State-of-the-art systems [4, 49] keep the base model stored in GPU memory and the adapters in host memory. Then, they fetch on-demand the adapters required by the running requests and discard them from the GPU memory as soon as the requests terminate. Some systems [49, 60] further fetch in advance the adapters for the requests waiting in the system's queue to hide some of the loading overheads. However, our study reveals that even such asynchronous adapter fetching increases the time-to-first-token (TTFT) latency, especially when the system is heavily loaded, as it increases contention in the CPU-GPU PCIe link.

Second, execution with adapters increases workload heterogeneity. This is because decoupled computations between base model and adapters increase the execution time of individual requests [60], and such effect varies across requests. Moreover, the use of adapters can increase resource utilization and throughput, which results in the execution of heterogeneous batches of requests for different tasks and adapters [4, 25, 49, 60]. With increased heterogeneity, tail latency is penalized: large requests that take long to execute end up stalling smaller requests within the same batch [25].

We analyze real-world production workloads [41] and observe that requests follow a heavy-tailed distribution: most are completed in a short time, while a small fraction experiences significantly longer execution durations. While prior work has largely attributed this heterogeneity to differences in input [59] and output [46] request sizes, our study is the first one to shed light on how the variability in adapter rank (i.e., size) [49] and popularity [10, 53, 60] affect the requests at the tail, underlying the necessity to take the adapter size into account.

Unfortunately, simply prioritizing short requests is insufficient to address the issue of tail latency. For instance, the speculative Shortest-Job-First (SJF) scheduler [46], along with its aging mechanism to mitigate starvation, inadvertently increases the tail latency of longer requests—potentially causing them to miss their Service Level Objectives (SLOs). Instead, our findings emphasize the need for a more nuanced scheduling strategy: one that addresses adapter-level heterogeneity, offers expedited processing for short requests, and ensures that longer requests still meet their SLOs.

We use these insights to design *Chameleon*, an LLM inference serving system optimized for many-adapter environments. Tasks share their base LLM, which uses a large fraction of the GPU memory, while each task uses its own specific adapter. Chameleon attains high efficiency through two new ideas.

First, Chameleon provides a *transparent, adaptive, and interference-free cache for adapters*. Contrary to common wisdom [4, 49], we observe that, even during high load, there is enough idle GPU memory to implement a cache for adapters that are likely to be reused in the future. However, as available memory fluctuates, the cache must be dynamically sized and carefully managed to avoid interfering with the key-value cache, while employing a cost-aware eviction policy suited for workload heterogeneity.

Second, Chameleon employs a *non-preemptive, adapter-aware multi-level queue (MLQ) scheduler* to minimize head-of-line blocking and ensure SLO compliance for all request types. Requests are classified into different queues based on their predicted sizes and, in each scheduling cycle, a subset from each queue is selected to form a batch. This enables a faster lane for smaller requests while also eliminating starvation across all request sizes.

We implement Chameleon on top of the open-source S-LoRA [49] LLM serving platform. Chameleon does not require any hardware or operating system support, or changes to CUDA kernels. We evaluate Chameleon with open-source LLMs using real-world production traces [41] and show that Chameleon is very effective. Compared to a state-of-the-art baseline [49], Chameleon reduces the P99 and P50 time-to-first-token (TTFT) latencies by 80.7% and 48.1%, respectively, while improving the throughput by 1.5×.

This paper makes the following contributions:

- A characterization of state-of-the-art LLM inference serving systems in environments with many LoRA adapters.
- The Chameleon LLM inference serving platform, which introduces the first cache design for LoRA adapters, and a novel adapter-aware multi-queue scheduler that eliminates head-of-line blocking while preventing starvation.
- An implementation and evaluation of Chameleon.

2 Background

LLM inference. Generative LLMs [29, 36, 47, 56] process the entire input at once (*prefill phase*) and then generate output tokens one by one (*decode phase*). In prefill, all input tokens are processed in parallel. This phase is compute-bound and its performance depends on the input size, which is known in advance. In decode, the output tokens are generated sequentially in iterations. Each iteration generates a token based on the input prompt and all previously generated tokens, typically cached on the GPU memory in *key-value (KV) caches*. The decode phase is memory-intensive and its performance depends on the output size, i.e. the number of decode iterations, which is determined on the fly and is unknown at the time a request is admitted to execute.

LLM inference serving systems. LLM serving systems batch requests for the same model to maximize hardware utilization. Since different requests generate different numbers of output tokens, the execution time of different requests in the same batch varies. To prevent long requests from blocking smaller ones, systems dynamically update batches. Specifically, state-of-the-art systems perform continuous batching [1, 61]: they remove completed requests from a batch and potentially add new ready-to-run requests on every decode iteration, an approach called iteration-level scheduling.

LLM Low-Rank Adaptation (LoRA). One way to reduce LLM training overhead is to pre-train LLMs and then fine-tune them for specific tasks. One method to do so is Low-Rank Adaptation (LoRA) [17, 43], where the layers of a base model are updated with low-rank matrices to fine tune them. These matrices are called LoRA adapters and their size (i.e., their *rank*) determines the accuracy of the resulting computation. Specifically, higher ranks potentially translate to better tuning and higher accuracy. Since different tasks have different accuracy requirements, they are likely to employ adapters of different ranks over the same base LLM [49, 58].

The straightforward way to apply adapters is to merge them with the base model and create a full-size standalone specialized LLM instance [28] for each task. However, recent works for LLM inference serving [4, 25, 49, 60] allow tasks to share the base model and allocate specific per-task adapters. Typically, an adapter is significantly smaller than the base model. Hence, this method significantly reduces the memory requirements of systems that serve diverse tasks [68]. Moreover, such method also enables batching requests of different tasks, with a common base and different adapter combinations, further improving the throughput.

Figure 1 shows the organization of such systems [4, 25, 49, 60]. On initialization, the base LLM model is transferred to the GPU memory from the host. A scheduler on the host manages the incoming requests, updating the batch to be executed on every iteration. Before it sends the batch to the inference engine on the GPU, the scheduler also loads any missing adapters required by the requests in the batch. Once there are no requests that use a given adapter, the adapter is discarded from the GPU memory to make room for new incoming requests [4, 49].

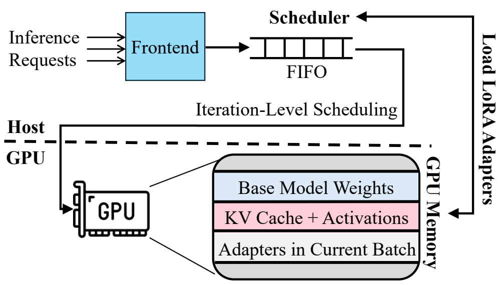

Figure 1: Conventional LoRA online serving system.

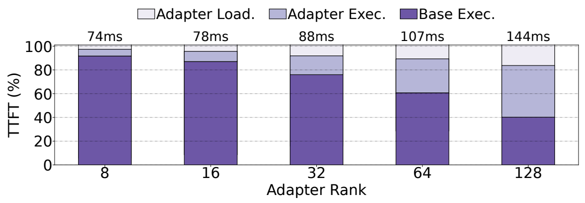

Figure 2: TTFT latency with different adapter ranks broken down into base and adapter execution, and adapter loading.

3 Opportunities in Many-Adapter Settings

In this section, we examine the new challenges that appear in many-adapter environments and why they are not efficiently handled by conventional LLM serving systems. We characterize the open-source Llama-7B model [56] on an NVIDIA A40 server with 48GB of GPU memory [38]. We use the S-LoRA serving platform [49], a state-of-the-art inference system for multi-adapter scenarios.

3.1 Adapters Increase Workload Heterogeneity

The LoRA adapters employed by different tasks are expected to vary in size (*rank*), as tasks require different levels of accuracy [17, 49, 58, 60]. Figure 2 shows how this rank heterogeneity affects the TTFT of a single inference request, with medium input and output size [53]. We run the request over a base Llama-7B model combined with a specific adapter on an unloaded system, and increase the adapter rank from 8 to 128 [49, 60]. We break down the total execution time into time spent: (1) executing the base model, (2) executing the adapter, and (3) loading the adapter's weights from host to the GPU memory. The numbers on top of the bars are the TTFT values.

We observe that, as the rank size increases, the relative weight of the adapter overheads also increases. For example, for rank 128, ~60% of the total TTFT latency is spent on adapter loading and computation.

For these experiments, we use the Multi-size Batched Gather Matrix-Matrix Multiplication (MBGMM) kernel from the state-of-the-art baseline system S-LoRA [49]. LoRA adapters induce two matrix multiplications on top of the base model multiplication, and a matrix addition for results aggregation per LLM inference layer. This leads to the high computational overhead of adapter execution observed in Figure 2. Recent work (Figure 5 in [60]) corroborates our findings that these steps are expensive even for small-rank adapters.

We further examine the effect of the adapter rank while considering other sources of inference heterogeneity. Prior work observed that large inputs lead to longer prefill phases and large outputs

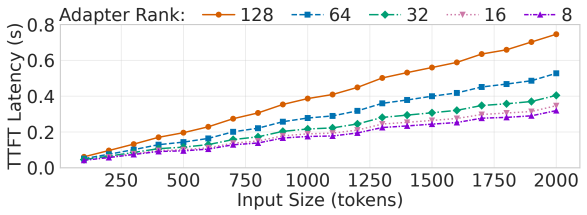

Figure 3: TTFT latency for different adapter ranks.

to much longer decode phases [53]. Also, using large batches of requests increases throughput but at the cost of longer decode iterations. Figure 3 shows the TTFT latency for different adapter ranks as we vary the input size of a request (i.e., the number of input tokens), while keeping the output size fixed. For this experiment, we keep the adapter weights in GPU memory and isolate prefill performance by excluding adapter loading. For all input sizes, TTFT varies significantly across adapter ranks. Moreover, the impact of the rank is more pronounced as the input size increases. Similarly, it can be shown that, for large batch sizes, different adapter ranks lead to diverse decode latencies for requests with similar input/output sizes. Overall, we find adapter rank to be an extra, equally important source of heterogeneity, next to input, output, and batch size.

Apart from different ranks, adapters have skewed popularity as well, following the skewed popularity of different tasks. LLM inference is a user-facing service where some tasks receive a larger amount of requests than others, and these requests typically arrive in bursts [10, 53, 60]. Next, we will show how this heterogeneity affects various system design decisions, such as which adapter to keep in GPU memory or how to schedule inference requests for different adapters.

Insight #1: Adapters are an additional source of heterogeneity in LLM inference that must be managed dynamically.

3.2 Adapters are Expensive to Load

When an LLM inference request using a specific adapter arrives at an online serving system, the adapter must be loaded into the GPU memory for the request to be processed. Thus, loading the adapter weights lies on the critical path of inference execution. Figure 2 shows that loading takes 17.5% of the total TTFT latency when a 128-rank adapter is used in an unloaded system.

This overhead becomes more pronounced as the requests use a larger number of different adapters. One reason is the contention on the PCIe link between the host and the GPU as the adapters are brought to the GPU memory. In our next experiment, we use rank 32 adapters and consider three scenarios: in *LoRA-1*, all the requests use the same adapter; in *LoRA-50* and *LoRA-500*, a request uses one of 50 or 500 different adapters with a uniform distribution. Figure 4 shows the normalized PCIe bandwidth consumption for the three scenarios and different requests per second (RPS). We normalize the bandwidth consumption to *LoRA-1* with 5 RPS.

We see that, as we go from *LoRA-1* to *LoRA-50* and *LoRA-500*, the bandwith consumption increases. With *LoRA-500*, the PCIe bus is saturated. We measure that at, 8 RPS, this bandwidth contention causes the P99 TTFT latency of the requests in *LoRA-50* and *LoRA-500* to be 1.69x and 2.60x higher, respectively, than *LoRA-1*. For higher RPS loads, these P99 TTFT latency gaps increase rapidly, but they are also affected by other bottlenecks.

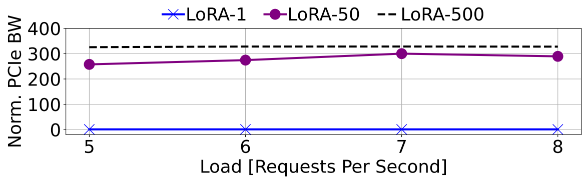

Figure 4: PCIe bandwidth usage under different loads for environments with: 1 adapter (*LoRA-1*), 50 different adapters (*LoRA-50*), and 500 different adapters (*LoRA-500*).

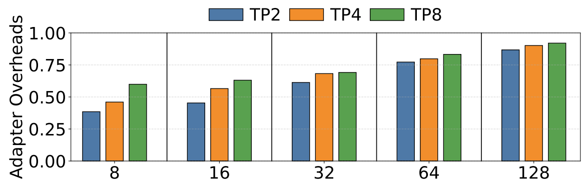

Figure 5: Overhead of loading the adapters of different ranks as a fraction of the total TTFT latency for Llama-70B running on 2, 4, or 8 A100 GPUs using tensor parallelism.

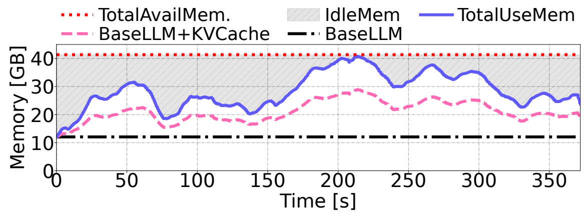

Figure 6: Memory usage over time for different parts of the workload: base LLM model, KV cache, and adapters.

We now evaluate the impact of adapter loading overhead when serving the larger Llama-70B base model using tensor parallelism (TP) across 2, 4, and 8 A100 GPUs—since the model no longer fits on a single GPU. We observe that the cost of loading adapters increases due to two main factors. First, larger base models result in proportionally larger adapter weight matrices, for the same rank configuration, increasing their loading time. For example, a rank 32 adapter for Llama-7B is 64 MB, while its size grows to 256 MB for Llama-70B. Rank 128 adapter size grows to the order of GBs. Second, using more GPUs introduces additional overheads: adapter weights must now be partitioned across tensor-parallel ranks, transferred separately to each GPU's memory, and synchronized to ensure consistent execution. These overheads exacerbate the latency on the critical path of inference.

Figure 5 considers different TP degrees and adapter ranks, and shows the fraction of the TTFT latency that is taken by adapter loading. We observe that this fraction increases with the TP degree and adapter rank. For example, loading accounts for 68% of the TTFT latency for rank 32 and TP4.

An intuitive way to reduce these overheads is to leverage idle GPU memory to cache adapters. However, LLM inference has substantial load fluctuations [53]. Figure 6 shows the GPU memory usage over time when we run the Llama-7B model using production traces of requests from Azure [41]. Because our testbed has modest memory, we have scaled down the input and output lengths in these large-scale system traces using a constant factor that results in the peak memory consumption of the scaled-down trace to be equal to the memory capacity of our testbed (§ 5.1).

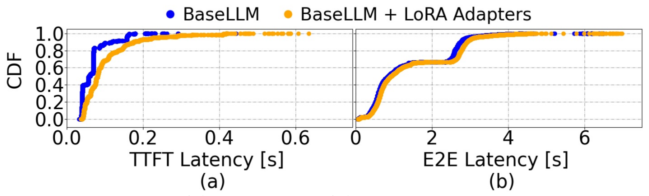

Figure 7: CDF of (a) TTFT and (b) E2E latency of requests for a real LLM trace [41]. Requests are executed one by one.

The figure shows the memory consumed by the base LLM (*BaseLLM*), base LLM plus KV cache (*BaseLLM+KVCache*), and base LLM plus KV cache plus adapter (*TotalUseMem*). We see that, most of the time, there is abundant idle memory that can cache adapter weights. However, idle memory drastically drops during load spikes. Hence, the system needs to carefully and dynamically resize the cache resources based on the incoming load.

These findings challenge the common design decision to discard the adapters from GPU memory if none of the currently running requests use them [49, 60]. We find that keeping them in GPU memory can significantly improve performance, especially in high load scenarios, and that there is a sizable amount of idle GPU memory that can be used for this.

Insight #2: Frequent loading of adapters from host to GPU memory creates bandwidth contention, degrading system performance. Idle GPU memory can be repurposed to cache adapters and mitigate some of these overheads. However, dynamic resizing of the cache is essential, as the amount of idle memory fluctuates heavily.

3.3 Adapters Affect Requests at the Tail

In Section 3.1, we observed that there is a high degree of heterogeneity in the performance of LLM inference requests, based on their input, output, and adapter size. Now, we analyze how this heterogeneity impacts the effectiveness of scheduling decisions.

We take the open-source production traces of LLM inference requests for a conversation service [41] and execute one request at a time. We run with only a base LLM, and with a base LLM augmented with LoRA adapters. Similar to [49], we consider a pool of 100 different adapters with rank sizes uniformly distributed among 8, 16, 32, 64, and 128, and associate every request in the trace with one of these adapters, following a uniform distribution for rank popularity and a power-law distribution for adapter popularity. Figure 7 shows the CDF of (a) TTFT and (b) end-to-end latency of all requests. For this experiment, the latency includes both the prefill phase and the time it takes to load the adapter. This figure shows that the execution time of requests follows a heavy-tail pattern: the majority of requests have short execution times, but there are a few very long requests. Moreover, adding LoRA adapters significantly affects requests at the tail.

Heterogeneity in execution times typically requires special scheduling considerations. LLM engines schedule requests at iteration-level [61] where, at each iteration, the scheduler decides which requests will execute in a batch. The majority of conventional systems use a FIFO approach due to its simplicity [20, 49]. However, FIFO is inefficient for heterogeneous requests, as it introduces head-of-line (HoL) blocking, leading to increased tail latency.

For this reason, researchers have proposed to schedule the requests in a Shortest-Job-First (SJF) manner. Specifically, existing

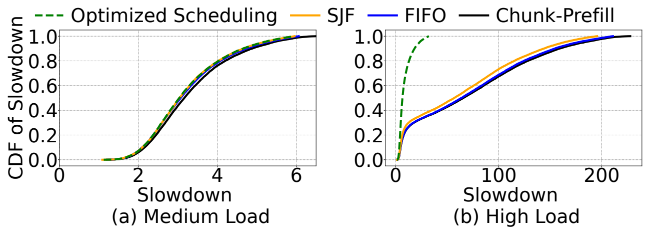

Figure 8: CDF of the per-request slowdown for different scheduling policies under (a) medium and (b) high load.

systems [46] predict the request output lengths and prioritize the requests with the shortest predicted outputs. However, continuously prioritizing short requests leads to the starvation of long requests, again, negatively impacting the overall tail latency. Moreover, using the output length as the only scheduling knob is insufficient, as inputs and adapters also impact the total latency (Figure 3).

To show the inefficiencies of these two scheduling policies, we execute the production trace [41] using the Llama-7B model. We record the slowdown of each request: how many times higher is the request's response time now relative to the response time in an isolated environment where the request executes alone. Figure 8 shows the CDF of slowdown per request with different scheduling policies: FIFO with regular iteration-level scheduling (i.e., continuous batching) [61] (*FIFO*); FIFO with the more advanced chunked-prefill iteration-level scheduling [1] (*Chunk-Prefill*); SJF; and the optimized scheduling policy that we will introduce in Section 4 (*Optimized Scheduling*). The last two schemes use iteration-level scheduling.

Under high loads, conventional policies create high slowdowns for the requests at the tail. In FIFO, short requests are blocked by long requests. Using a classification of requests into short, medium, and long that we describe in Section 4, we measure that a short request spends on average 28.6% of its time waiting to be scheduled, compared to 12% for a large request. As chunked-prefill is designed to prioritize decode iterations, it slightly slows down prefill iterations, increasing TTFT latencies. Chunked-prefill does not solve the HoL blocking problem because it still adheres to a per-request ordering within each pipeline stage. Thus, short requests can remain blocked behind long prefill or decode chunks in their respective queues, especially when resources (e.g., tokens or compute slots) are saturated. Hence, chunked-prefill does not reduce tail latency under high-load scenarios.

For SJF, long requests are penalized, as they are starved by the prioritization of short ones. We measure that a long request spends 5.15 s waiting to be scheduled compared to 1.5 s for a small request. While long requests are relatively infrequent, their queuing latency has a significant impact.

Insight #3: Conventional scheduling policies such as FIFO and SJF are ineffective for highly heterogeneous LLM inference requests. There is a need for a scheduling policy that can efficiently manage request heterogeneity while, at the same time, taking into account all knobs that affect the execution time.

4 Chameleon Design

4.1 Overview

Based on all the previous insights, we design *Chameleon*, an LLM inference serving system optimized for many-adapter environments.

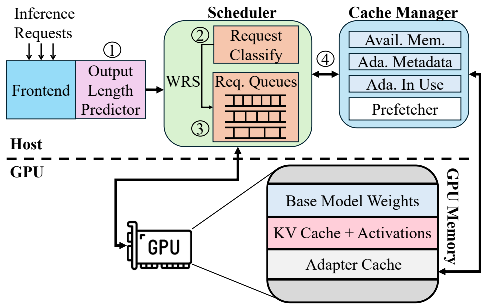

Figure 9: Chameleon architecture.

Chameleon is designed to address the unique challenges posed by these environments: (1) the overheads and side-effects of loading the adapters' weights and (2) the increased request tail latency due to inefficient request scheduling.

To address the first issue, Chameleon uses underutilized GPU memory to implement a software-managed adapter cache. This *Chameleon Adapter Cache* stores adapter weights in GPU memory, removing adapter loading off the critical path, and reducing PCIe bandwidth usage. To address the second issue, Chameleon uses a multi-queue non-preemptive scheduler that provides an express lane for short requests, while ensuring that no request starves. This *Chameleon Scheduler* works synergistically with the Adapter Cache, maximizing system throughput.

Figure 9 overviews the Chameleon architecture. To handle workload heterogeneity, Chameleon classifies all incoming requests based on their total size. Every request first goes through an output length predictor that estimates the number of output tokens ①. We use an existing, open-source, predictor based on a BERT proxy model [46]. Then, the Chameleon scheduler combines this estimated output size with the known number of input tokens and the rank of the adapter required by the request to calculate a *Weighted Request Size (WRS)* ②.

Chameleon uses the WRS to categorize requests into classes, e.g., *small*, *medium*, and *large*, and admit them to different request queues ③. Chameleon uses iteration-level scheduling (Section 2) and, therefore, on every decode iteration, it can remove and add requests to a batch. Importantly, on every iteration, Chameleon tries to take requests from all queues, while respecting the queues' assigned quotas. Specifically, each queue is assigned some amount of GPU resources that the requests from that queue can consume at each iteration. The scheduler creates a fast lane for short requests preventing their HoL blocking, but still admits requests from all queues guaranteeing that no request will starve. To handle load fluctuations and changes in request properties, Chameleon dynamically adjusts both the number of queues and the per-queue cutoffs based on the monitored WRS distribution of the incoming load.

On every scheduling decision, Chameleon also invokes its *Cache Manager*. This is a software controller that manages the Chameleon Adapter Cache. Its function is to: (i) (pre)fetch any necessary adapters required by the requests to be scheduled and (ii) evict any idling adapters when the GPU memory does not have room to store all the necessary state for the incoming requests ④. The Cache Manager tracks all cached adapters and the necessary metadata to enforce a cost-aware eviction policy.

4.2 Chameleon Adapter Cache

The Chameleon Adapter Cache (or Chameleon Cache, for short) is a software structure that stores unused adapters loaded by previous requests in idling GPU memory. The goal is to eliminate the costs of fetching the adapters again in the future, on the critical path of an inference request. Chameleon maintains one cache instance per LLM instance, i.e., each LLM replica has its own local adapter cache. Each cache entry contains the adapter's weights and some metadata used for cache management. The metadata contains:

- **Adapter ID**: A unique identifier for the adapter.
- **Adapter Rank**: The size of the adapter, which affects the amount of GPU memory it occupies.
- **Last Used Timestamp**: The last time the adapter was accessed.
- **Usage Frequency**: The total number of times the adapter has been used within a specific time frame.
- **Reference Counter (RC)**: The number of active requests using this adapter. If RC is zero, the adapter is eligible for eviction.

The cache is managed by a Cache Manager. The manager performs the following operations: (i) it retrieves a cached adapter for an incoming request, (ii) it loads a missing adapter from host memory, (iii) it dynamically resizes the cache based on the incoming load, and (iv) it employs a cost-aware eviction policy to discard cached adapters when necessary. Since adapter weights are read only, there is no need to be concerned about their coherence, or to write them back to the host memory on eviction from the cache. Next, we describe three aspects of the Chameleon Cache.

1. Dynamic Cache Sizing. Unlike in hardware caches, the capacity of the Chameleon Cache changes dynamically over time. The Cache Manager adjusts its size in real time, to ensure that the resource demands of incoming requests are always met (Figure 6). For example, when the input activations, KV entries, or missing adapters of incoming requests would not fit in the available free GPU memory, the Cache Manager downsizes the Chameleon Cache, evicting adapters to free up the necessary space. Similarly, when a request ends and there is enough idling GPU memory, the Cache Manager expands the Chameleon Cache to store the request's adapter.

The Chameleon Cache Manager and Scheduler work synergistically for cache resizing. As the Scheduler scans the request queues and assembles a batch of requests to be scheduled on every decode iteration (Section 4.3), it monitors the requests' memory requirements. It then communicates the exact amount of memory required by the batch to the Cache Manager. The latter, if necessary, discards unused adapters to free up space, based on a cost-aware eviction policy, and loads any missing adapters from the CPU memory.

2. Cost-Aware Eviction Policy. The cache eviction policy in our multi-adapter environment needs to be more sophisticated than existing policies based on recency like least recently used (LRU). While these policies can capture the temporal locality in the adapter requests, they may fail to capture the adapters' skewed popularity found in LLM serving workloads. Indeed, certain adapters are accessed more frequently than others, or are used by a larger number of concurrent inference requests [49]. Evicting frequently-used adapters can increase miss rates and the CPU-GPU link bandwidth consumption due to frequent adapter reloading.

An additional reason why LRU is insufficient is that cache misses in this environment have varying costs. Unlike hardware caches, which store cache lines of fixed size, the Chameleon Cache caches adapters, which have different sizes, i.e., ranks. Consequently, the latency to load an adapter on a cache miss is not fixed. Larger adapters take longer to transfer from host to GPU memory. Thus, the eviction policy must be *cost-aware* [12] and prioritize the eviction of smaller adapters.

Similar to caching schemes employed in other domains [12], we show that relying solely on a single feature is insufficient to capture the complex trade-offs in our system. To address this limitation, we propose a *compound eviction algorithm* that considers multiple factors influencing adapter importance. Our scheme calculates a *score* for each adapter based on its frequency of use, recency of access, and size. Frequency of use is significant, as some adapters are more popular than others. Recency is important for temporal locality, as bursts of requests for the same model/adapter are a common access pattern [41]. The size of the adapter affects the cost of a cache miss, as larger adapters are costlier to reload. We combine these factors linearly to calculate an eviction score: $Score = F \times Frequency + R \times Recency + S \times Size$, where F, R, and S are weighting coefficients. The adapter with the lowest score is considered the least critical and is evicted first.

The weighting coefficients F, R, and S enable adjusting the sensitivity of the eviction policy to each factor. For example, if frequency is deemed more important than recency for the running workload, F can be assigned a higher value relative to R. For this study, we use static coefficients, which we set by offline profiling of industrial traces of inference requests [41] combined with adapter size distributions found in the literature [49]. F, R, and S are set to 0.45, 0.10, and 0.45, respectively.

Chameleon never evicts adapters that are actively used by running requests. To guarantee this, the Cache Manager maintains a reference counter per adapter that tells how many requests are currently using the adapter. It considers eligible for eviction only the adapters whose counters have dropped to zero. Additionally, the Manager checks the Scheduler to identify adapters associated with queued requests. Such requests are not currently running, but are guaranteed to execute in the near future. The Manager attempts to retain these adapters in the cache, provided there is sufficient memory available. Overall, the Manager applies the eviction policy only to adapters that are not used by both currently-running and queued requests. The adapters of queued requests are considered for eviction only when memory constraints make it necessary.

3. Prefetching. Building on prior-art optimizations, Chameleon monitors the request queues and, whenever possible, prefetches the missing adapters required by the waiting requests before they are admitted to a batch for execution [49, 60]. This approach can lead to late prefetching, where the request becomes ready for execution before the prefetching is completed. Hence, we explore techniques that predict future load, such as a histogram-based approach [48], and prefetch adapters even for requests that are not currently queued. Since the effectiveness of this optimization depends heavily on the prediction accuracy, we do not enable it by default in our evaluation; we include a separate experiment to assess the potential impact of such prefetching.

Prefetching has the potential to hide the cost of loading an adapter's weights and remove it from the request's critical path of execution. However, prefetching still consumes CPU-GPU link bandwidth. Thus, caching the adapters is still necessary for high performance.

4.3 Chameleon Scheduler

The Chameleon Scheduler is inspired by prior work on load balancing multi-server environments with heterogeneous task size distributions [7, 15, 35]. The scheduler stores the inference requests across multiple queue lanes, each dedicated to handling requests within a specific size range. Its goal is twofold: to provide a fast lane for small requests, preventing their HoL blocking, and to ensure that requests from all lanes are scheduled in parallel, avoiding starvation for large requests.

Each queue is assigned a resource quota, which governs the resources available to execute requests from that queue. This quota is represented as tokens, and includes input tokens, output tokens, and tokens due to the memory required for the corresponding adapter. These tokens determine the resources a queue can reserve for request execution. When a request from a specific queue is admitted to the batch, the queue's available quota is decreased by the memory consumption of the request, determined by its input and output length, and adapter size, all translated into tokens. When the request ends, it returns the borrowed quota back to the queue. In every iteration, all queues have the chance to put requests into the batch, although the queues with smaller requests are accessed first.

1. Admission to the Queues. We characterize the requests entering the system by three parameters: known input size, predicted output size, and rank of the used adapter. We calculate the weighted request size (WRS) as an estimate of the total execution time of a request based on the formula:

$$\text{WRS} = \left(A \cdot \frac{\text{InputSize}}{\text{MaxInputSize}} + B \cdot \frac{\text{OutputSize}}{\text{MaxOutputSize}}\right) \cdot \frac{\text{AdapterSize}}{\text{MaxAdapterSize}}$$

The input size affects the prefill latency, which is typically shorter than the decode latency but still a significant contributor to the total execution. The output size determines the number of decode iterations, which affects the decode latency and the total execution time. The adapter size affects the speed of both prefill and decode processes (Section 2). It can be shown that using this polynomial of degree 2 improves Chameleon's performance by up to 10% over using a polynomial of degree 1 that simply combines the three factors linearly.

We call WRS the "request size", and use it to classify requests into size ranges and dispatch them to corresponding queues for scheduling. A and B are weighting coefficients chosen based on our sensitivity studies and on profiling in Section 3. We set A to 0.4 and B to 0.6.

Given a request, the scheduler uses the calculated WRS and the per-queue cut-offs (i.e., the boundaries that define the ranges of request sizes for each queue) to place the request in the correct queue. Later, we detail how to determine these per-queue cut-offs using request clustering. Note that the Chameleon Scheduler uses an open-source BERT-based proxy model to predict a request's output length [46].

2. Admission to the Batch. The idea behind the Chameleon Scheduler is depicted in Algorithm 1. It operates in two phases: *Initial Request Admission* and *Redistribution of Spare Resources*. In the first phase, each queue attempts to put requests into the batch, up to the queue's maximum allowed resources. If certain queues have few or no requests to put, any unused resources are collected and consolidated into a *Total Spare Resources* bucket. After the first phase ends, in the second phase, the scheduler redistributes the spare resources to queues that still have pending requests, aiming to maximize resource utilization. Specifically, starting from the smallest-request queue and moving downward to larger-request queues, the scheduler allocates as much of the spare resources as possible to admit waiting requests into the batch. If requests from a given queue still cannot be admitted due to insufficient tokens available, no additional resources are allocated to that queue.

The phases of this process are illustrated in Figure 10, which depicts three request queues, for "small", "medium", and "large" requests. Figure 10(a) shows the case when no spare resources are collected, while Figure 10(b) shows the case when spare resources are collected and redistributed. In Figure 10(a), the Initial Request Admission phase starts with the small-request queue, admitting three requests, which fit within the queue's resource quota (1a). The fourth request is not admitted due to insufficient resources allocated to the queue. These admitted requests are then placed into the batch (1b). The same procedure is subsequently applied to the medium-request queue (2), and the large-request queue (3). At the end of

Algorithm 1: Generate a new batch of requests.

```
def generate_batch:
    Inputs: Queues = requ. queues; PQ_Tokens = per queue tokens
    Result: Batch of requests to be sent to the GPU.
    batch ⟵ [];
    leftover ⟵ 0;
    for each q in Queues do // Phase 1
        consumed ⟵ put_batch(q, PQ_Tokens[q], batch);
        if q is empty then
            leftover ⟵ leftover + (PQ_Tokens[q] - consumed);
    for each q in Queues do // Phase 2
        if leftover == 0 then
            break;
        consumed ⟵ put_batch(q, leftover, batch);
        leftover ⟵ leftover - consumed;
    return batch;

def put_batch:
    Inputs: Queue; Tokens; Batch
    Result: Tokens consumed by added requests from the queue.
    resources ⟵ Tokens;
    consumed ⟵ 0;
    for each req in Queue do
        needed ⟵ need_resources(req);
        if resources < needed then
            break;
        resources ⟵ resources - needed;
        consumed ⟵ consumed + needed;
        batch.append(req);
    queue ⟵ [req for each req in queue if req not in batch]
     return consumed;
```

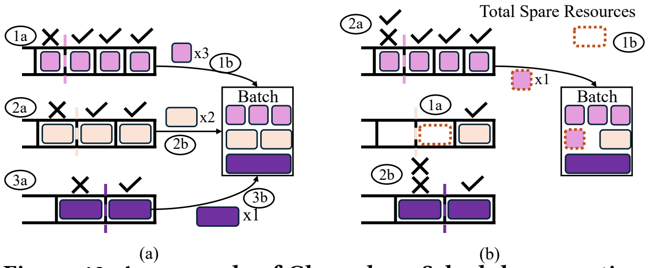

Figure 10: An example of Chameleon Scheduler operation when (a) no spare resources are collected (b) spare resources are collected and redistributed.

this phase, there are no remaining resources to redistribute, so the process concludes without entering the second phase.

Figure 10(b) shows a case where spare resources are available for redistribution. Specifically, the medium-request queue only has a single request, and so it does not use up its allocated resources. Hence, during the Initial Request Admission phase, the queue contributes with some spare resources (1a), which are deposited into the Total Spare Resources bucket (1b).

Then, during the Redistribution of Spare Resources phase, the scheduler checks each queue in order, from the small- to the medium- and large-request queue, to try to admit any remaining requests into the batch. The small-request queue evaluates if its single pending request can be admitted (2a). Since the Total Spare Resources are sufficient, the scheduler allows this request to be admitted. The medium-request queue has no pending requests and is skipped. The large-request queue attempts to put its pending request (2b). However, since the remaining spare resources are not enough, the request remains in the queue. At the conclusion of this phase, the batch is finalized and ready for execution.

3. Opportunistic Bypassing. Sometimes, a request *R1* that should be put in a batch according to the resource quota allocated to its queue, may fail to get admitted to the batch because there is not enough idle GPU memory to store its adapter—even after Chameleon evicts all idle cached adapters. In such case, without proper action, the queue is unable to use its allocated resource quota. However, it may happen that a younger request *R2* in the same queue uses an adapter that is either already loaded in the Chameleon Adapter Cache or is small enough to fit in the remaining space of the cache.

To address this case, Chameleon implements an *Opportunistic Bypass* mechanism, whereby *R2* is put in the batch for execution, bypassing *R1*. This mechanism improves system throughput by allowing more requests to be processed without waiting for cache space to become available. However, repeated bypassing can lead to request starvation. Consequently, Chameleon first predicts how soon will the memory needed by *R1* become available, and how long will *R2*'s execution take. Then, Chameleon allows *R2* to bypass *R1* only if the former is longer than the latter.

Unfortunately, predictions may turn out to be wrong. Hence, if, before *R2*'s execution completes, Chameleon finds enough free memory on the GPU (including the memory used by *R2*) to execute *R1*, it squashes *R2* for later re-execution. In our experiments, we see at most 5% of requests getting squashed. Note that these scenarios can only happen when the GPU memory is entirely consumed by running requests and no idle adapters are cached. Thus, eviction policies do not apply here.

4. Determining the Number of Queues. The efficiency of the Chameleon Scheduler depends on the number of used queues. Too few queues may cause HoL blocking when there is a high variability in the request sizes within a queue, while too many queues can result in load imbalance and underutilized queue resources due to resource fragmentation. To decide on the optimal number of queues, the Chameleon Scheduler uses *K-Means clustering*. Given the distribution of request sizes, the scheduler computes K-Means clustering for values of K ranging from 1 to K_{max}. With K-Means clustering, requests similar in size are grouped within the same cluster, and requests from different clusters are different enough to require separate resources. For each value of K, the scheduler calculates the Within-Cluster Sum of Squares (WCSS), and picks the K that yields minimal WCSS as the optimal number of queues. We set the maximum number of queues, K_{max}, to 4 to keep queue management overheads tolerable.

Once we have the K centroids from the clustering result, we proceed to determine the per-queue request-size cutoffs. Specifically, we define the cluster boundaries as the midpoint between the centroids of two consecutive clusters. For example, the boundary between $Cluster_i$ and $Cluster_{i+1}$ is $(Centroid_i + Centroid_{i+1})/2$. The boundaries represent the maximum and minimum request sizes for each queue: $Queue_1$ handles requests smaller than $Boundary_1$, $Queue_2$ handles requests larger than or equal to $Boundary_1$ but smaller than $Boundary_2$, and so on for all K queues.

The distribution of request sizes changes over time due to fluctuating load behavior. Hence, static queue configurations can lead to inefficiencies. Therefore, Chameleon dynamically adjusts the number of queues based on the observed load patterns. Specifically, the system periodically gathers recent request data to analyze the distribution of request sizes and, every $T_{refresh}$, re-computes the optimal number of queues and the per-queue cut-offs using the aforementioned method. Since changes in load patterns are not sharp [53], changing the multi-queue organization happens relatively infrequently. We set $T_{refresh}$ to 5 minutes, which adds negligible overheads.

5. Assigning Quotas per Queue. After determining the number of queues in the system and the per-queue cut-offs, the Chameleon Scheduler assigns the resource quotas to each queue. For this, we use queuing theory, modeling the system as $K * M/M/1$ queues [34].

We take the maximum allowed size (S) of a request in a queue in tokens, the assigned resource quota (Tok) to the queue in tokens, the expected time duration (D) of processing a request from the queue, the arrival rate (λ) of requests to the queue, and the requests' SLO. Then, the processing rate of the requests is $\mu = \frac{Tok}{S*D}$, while the total time that a request spends in the system is $T_{total} = \frac{1}{\mu - \lambda}$. To meet the SLO, the system needs to satisfy the following equation: $T_{total} \leq SLO$. Combining these constraints, we compute the minimum assigned quota in tokens (Tok_{min}) to the queue that is required for requests from the queue to meet the SLO:

$$Tok_{min} \geq S * D * \left(\frac{1}{SLO} + \lambda \right)$$

The total number of available tokens in the system (Tok_{total}) must be greater than or equal to $\sum_q Tok^q_{min}$ (i.e., the sum of the minimum number of tokens needed by each queue q). Then, each queue q is assigned its minimum number of required tokens (Tok^q_{min}), and the remaining tokens (Tok_{total} - $\sum_q Tok^q_{min}$) are split across queues proportionally to their initial weights.

To adjust to the dynamic nature of the workload, Chameleon recomputes the per-queue quotas every $T_{refresh}$.

4.4 Multi-GPU Set-up

With multiple GPUs, LLM inference can use tensor parallelism (TP), pipeline parallelism (PP), and data parallelism (DP). In TP or PP, Chameleon distributes its adapter cache accordingly, so each GPU stores a fraction of each adapter; in DP, Chameleon replicates the adapter cache across engines. Since adapters are read-only, data coherence is not a concern. In this paper, we follow the S-LoRA TP strategy [49].

For scheduling, in TP or PP, Chameleon treats all GPUs as a single execution engine; in DP, Chameleon uses a two-level scheduler: a global scheduler dispatches requests to the different engines, and each engine has its local scheduler.

5 Evaluation

5.1 Evaluation Methodology

Hardware Platforms and LLMs. We run most of our experiments on a server equipped with an A40 NVIDIA GPU [38] and an AMD EPYC 9454 CPU. The GPU has 48GB memory and the CPU has 48 cores and 377GB of main memory. For the scalability experiments, we use a server equipped with an A100 NVIDIA GPU [37] configured with 24GB, 48GB, and 80GB of GPU memory. For the multi-GPU experiments, we use four A100 GPUs with 80GB of GPU memory. For the majority of experiments, we use the Llama-7B [56] model. When memory capacity allows, we also run the Llama-13B and Llama-30B models. We used other models, such as Falcon [55], OPT [30], and Mixtral [36] and observed similar trends.

Workload Configuration. We set the input and output lengths of requests based on the open-source production trace from Azure [41]. To vary the load on the system, we use the Poisson distribution for the request inter-arrival time [4, 25, 26]. We set the number of different adapters used by the requests to N_a. Unless specified otherwise, in our experiments, N_a is 100. There are five adapter ranks: 8, 16, 32, 64, and 128. Each rank has an equal number of different adapters, i.e., $N_a/5$. To each request, we attach an adapter, following a uniform distribution for rank popularity and a power-law distribution for adapter popularity within a rank [49].

Baseline Systems. We run the experiments on S-LoRA [49], an open-source state-of-the-art LLM inference serving platform for adapter environments, and compare Chameleon to the baseline S-LoRA. S-LoRA performs iteration-level scheduling using a FIFO policy, and asynchronous adapter prefetching without adapter caching. We also compare Chameleon's scheduler to the recently proposed SJF scheduler in μServe [46]. We measure Time-To-First-Token (TTFT), Time-Between-Tokens (TBT), and End-To-End (E2E) latency. We set the SLO to be 5× the average request execution time in a low-load system [26, 41, 53].

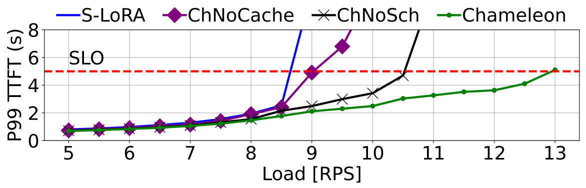

Figure 11: P99 TTFT tail latency for *S-LoRA*, *ChameleonNoCache*, *ChameleonNoSched*, and *Chameleon* under different loads. The red dashed line indicates the SLO.

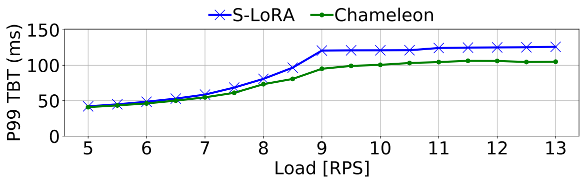

Figure 12: P99 TBT tail latency for *S-LoRA* and *Chameleon*.

5.2 Performance Gains

1. Tail Latency. Figure 11 shows, for different loads, the P99 TTFT tail latency for *S-LoRA*, *Chameleon* without its cache, *Chameleon* without its scheduler, and the full *Chameleon*. We consider the latter in this section. Although it is hard to see for low RPS, *Chameleon* consistently has lower TTFTs than *S-LoRA*, and the benefits become more pronounced as the load increases. At low (6 RPS), medium (8 RPS), and high (9 RPS) loads, *Chameleon* reduces the TTFT tail latency over *S-LoRA* by 14.7%, 24.6%, and 80.7%, respectively.

There are two reasons why *Chameleon* reduces the P99 TTFT latency over *S-LoRA*. First, its caching mechanism reduces the adapter fetching time and, as the load increases, it also alleviates the PCIe bandwidth bottleneck—which in turn further decreases TTFT latency. Second, its scheduling policy reduces queueing delays, as it removes HoL blocking and prevents starvation, especially helping the requests at the tail. As the load increases, GPU memory is increasingly consumed by the KV cache entries of the running requests, and there is less space for Chameleon to cache adapters not currently in use. It can be shown that, by 12.5 RPS, most of the time, GPU memory is fully used and there is no space for caching adapters not in use. Still, we see that Chameleon manages to reach this point while keeping TTFT under SLO. Below 12.5 RPS, Chameleon judiciously re-purposes scarce idle memory, caching frequently-used and costly to reload adapters, while prioritizing requests with short execution times that use them. S-LoRA, on the other hand, already violates SLO at about 8.5 RPS, well before it can fully utilize all the available GPU memory to run requests.

Chameleon reduces both TTFT and TBT tail latencies. Figure 12 shows the P99 TBT latency for *S-LoRA* and *Chameleon* under different loads. Again, *Chameleon* has lower latencies than *S-LoRA* for all loads. However, both systems keep their TBT latency under the SLO (150ms). The reason is that TBT latencies are less affected by queuing effects, and requests do not wait on adapter loading. Substantially increasing the batch size can in theory increase TBT in both systems. However, in our experiments of Figure 12, we find that admissions are eventually limited by available GPU memory, and batches never grow to the extent of violating TBT SLOs.

2. Throughput. Figure 11 also shows the expected TTFT SLO as a red dashed line (i.e., 5× the average request execution time in

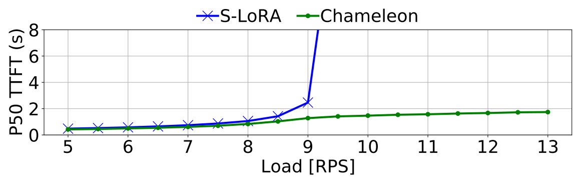

Figure 13: P50 TTFT latency for *S-LoRA* and *Chameleon*.

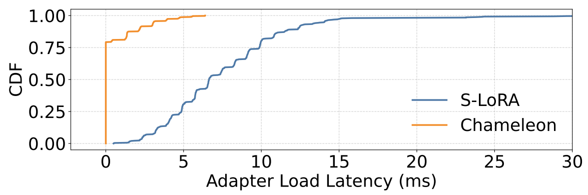

Figure 14: CDF of adapter loading latency on the critical path.

a low-load system). We define the throughput as the load that a system can sustain without violating this SLO. From Figure 11, we can see that *S-LoRA*'s starts violating the SLO around 8.6 RPS, while *Chameleon*'s starts violating the SLO around 12.9 RPS. This results in 1.5× higher throughput for *Chameleon*.

3. Median Latency. Figure 13 shows the P50 TTFT latency for *S-LoRA* and *Chameleon* under different loads. At low (6 RPS), medium (8 RPS), and high (9 RPS) loads, *Chameleon* reduces the median latency over *S-LoRA* by 13.9%, 20.9%, and 48.1%, respectively. The benefits of *Chameleon* are still significant, although not as pronounced as in the tail latency. The reason for this is that average conditions are less demanding.

4. Performance Breakdown. To understand the performance benefits of the two main *Chameleon* techniques, we run them in isolation. Figure 11 shows the P99 TTFT latency of *Chameleon* when running only with either our caching technique (*ChameleonNoSched*) or our scheduling technique (*ChameleonNoCache*). We see that both systems improve the throughput over *S-LoRA*: *ChameleonNoSched* and *ChameleonNoCache* have 1.2× and 1.05× higher throughput, respectively, than *S-LoRA*. However, their throughput is substantially lower than *Chameleon*'s. Hence, both adapter caching and adapter-aware scheduling are needed.

5. Adapter Loading Time. Figure 14 shows the CDF of the latency of adapter loading on the critical path for the requests of the Splitwise trace [41] in *Chameleon* and *S-LoRA*. *S-LoRA* suffers from adapter loading latencies of up to 30ms, as its prefetching scheme fails to completely overlap adapter transfer with computation. With *Chameleon*, on the other hand, 75% of the requests hit in the Chameleon Cache, resulting in zero loading overheads, while the remaining 25% of the requests pay loading costs of only up to 6ms. Adapter loading in *Chameleon* is cheaper because: a) *Chameleon* prioritizes the eviction of smaller adapters and thus reloading on a cache miss is cheaper, and b) *Chameleon*'s caching reduces the contention on the PCIe.

5.3 Different Scheduling/Caching Policies

In earlier experiments, we compared Chameleon with S-LoRA, which performs FIFO request scheduling and does not cache unused adapters. Here, we compare Chameleon to a SJF (shortest-job-first) scheduling policy proposed by μServe [46]. Also, we augment the baseline with a Chameleon Cache that uses an LRU eviction policy.

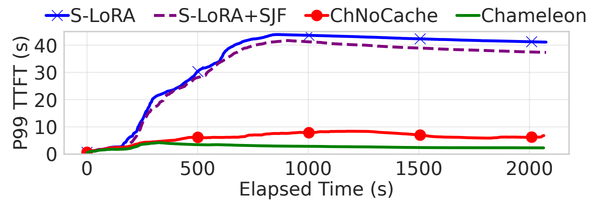

Figure 15: P99 TTFT latency over time with different scheduling policies: FIFO (default in *S-LoRA*), SJF in *S-LoRA*, and our proposed policy in *ChameleonNoCache* and *Chameleon*.

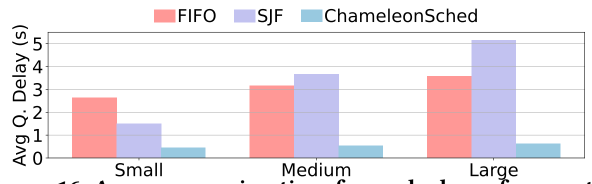

Figure 16: Average queuing time for each class of request in *S-LoRA's FIFO, SJF,* and the *Chameleon* Scheduler.

1. Scheduling Policies. Figure 15 shows the P99 TTFT latency over time with different scheduling policies driven by the production traces in Spitwise [41] at 9 RPS. We run *S-LoRA* with its default FIFO scheduling policy [49] and *S-LoRA* with the SJF scheduling policy from *μServe* [46], as two state-of-the-art baselines. Additionally, we run our proposed adapter-aware multi-queue scheduling policy, both without our caching mechanism (*ChameleonNoCache*) and with it (*Chameleon*).

Both *S-LoRA* and *S-LoRA*+SJF have large tail latencies that increase over time due to the queuing bottlenecks. Their TTFT latencies amply violate the SLO. With FIFO scheduling (*S-LoRA*), the requests at the tail are short ones blocked by the earlier long ones, while with SJF scheduling (*S-LoRA+SJF*), the requests at the tail are long ones starved by the prioritization of short requests. Our proposed scheduling policy (*ChameleonNoCache*) is very effective: it removes both HoL blocking effects and starvation, leading to much lower tail latencies. Finally, by integrating our caching approach, the TTFT latency reduces further.

2. Characterizing the Scheduling Policies. To understand why the Chameleon Scheduler outperforms the other schedulers, we measure the time that requests spent waiting in the queues before they are served. In Figure 16 we plot the average queuing delays per request size category, as identified by *Chameleon* (small, medium, and large), and for the three scheduling policies, i.e. *S-LoRA's FIFO, SJF*, and the *Chameleon* Scheduler. We see that FIFO introduces relatively uniform absolute queuing delays. However, for small requests, queuing delays account for 28.6% of their E2E latency. On the other hand, the SJF scheduler prioritizes small requests, creating long queuing delays for large requests. Finally, the Chameleon scheduler substantially reduces queuing delays for all request types, bringing delays to below 8% of the requests' E2E for all sizes.

3. Caching Policies. We now compare different replacement policies for our proposed adapter cache. Specifically, *LRU* evicts from the cache the least recently used adapter. *FairShare* follows our proposed approach of considering the adapter's recency, frequency, and size, but assigns the same weight to all three knobs. Finally, *Chameleon* uses our proposed algorithm, where the weights of the three knobs are tuned based on our extensive profiling (Section 4.2).

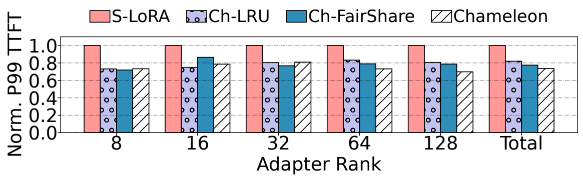

Figure 17: Normalized P99 TTFT latency for *S-LoRA*, *Chameleon-LRU*, *Chameleon-FairShare*, and *Chameleon*.

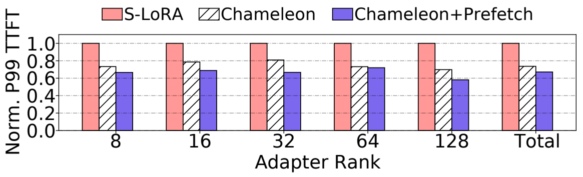

Figure 18: Normalized P99 TTFT latency for requests of different adapter ranks in *S-LoRA*, *Chameleon*, and *Chameleon+Prefetch*.

Figure 17 shows the normalized P99 TTFT latency for requests of different adapter ranks at medium system load (8 RPS) for *S-LoRA* (which does not have an adapter cache) and for Chameleon with the three adapter cache replacement policies described above. We see that Chameleon's proposed caching mechanism is very effective. All the caching schemes reduce the P99 TTFT latency over *S-LoRA* by a considerable amount for all adapter ranks. Additionally, our proposed replacement policy further reduces the TTFT, especially for larger adapters. For example, for requests with adapter rank 128, *Chameleon* reduces the P99 TTFT latency over *Ch-FairShare* by 12%. For the total trace, *Ch-LRU*, *Ch-FairShare*, and *Chameleon* reduce the P99 TTFT latency over *S-LoRA* by 18%, 22%, and 26%, respectively.

Chameleon's eviction policy is based on cost and benefit estimations. Prior work on software caches for objects with variable sizes proposed the Greedy Dual Size Frequency (GDSF) algorithm for web caching [5]. GDSF uses $Score = Frequency * Cost/Size + K$ to identify eviction candidates, where Cost is the overhead to load an object into the cache. Chameleon applies this strategy to the new context of adapter caching, and proposes a new score formula for this use-case. GDSF's score is sub-optimal for a) the skewed access patterns of adapters, as it tends to cache only the most popular adapters and discards the rest, and b) the skewed rank popularity of adapters, as GDSF aggressively evicts larger adapters with moderate use frequency. It can be shown that the P99 TTFT for high load (9.5 RPS) and power-law adapter popularity for *S-LoRA* with the cache and eviction algorithm of GDSF, is substantially worse than that of Chameleon.

4. Prefetching Mechanism. To reduce the latency of cache misses, Chameleon could use prefetching. It could predict which adapters are going to be used in the near future, and prefetch them to the adapter cache ahead of time. To test this idea, we have used a histogram-based technique to predict the future load of requests from [48]. In this section, we show the potential benefits of using this prefetching. However, since the effectiveness of prefetching is highly dependent on the prediction accuracy of future loads, we do not include prefetching by default in our experiments in this paper.

Figure 18 shows the normalized P99 TTFT latency for requests of different adapter ranks under medium load in three systems: *S-LoRA*, *Chameleon*, and *Chameleon+Prefetch*. We see that prefetching can further reduce the TTFT latency of *Chameleon*. For the total trace, prefetching further reduces the P99 TTFT latency by 8.8%. As adapters are set to follow a uniform distribution for rank popularity and a power-law distribution for adapter popularity within a rank, their predictability is high. However, for other distributions, predictability may be lower.

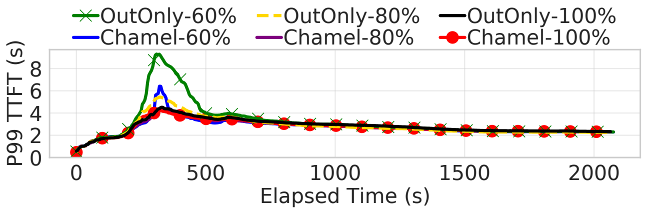

Figure 19: P99 TTFT latency over time for two configurations (*OutputOnly* and *Chameleon*) under different output length predictor accuracies.

5.4 Sensitivity Analysis

To gain further insights into Chameleon, we perform a sensitivity analysis of several of its parameters.

1. Impact of the Accuracy of the Output Length Predictor. Recall that the Chameleon Scheduler uses an open-source BERT-based proxy model to predict a request's output length (Section 4.3.1). We measure that our predictor has an average accuracy of about 80%. In this section, we examine the impact of artificially setting the predictor accuracy to 100%, 80%, and 60%. We consider two ways to compute the weighted request size (WRS) (Section 4.3.1): *OutputOnly*, which uses only the request output length (similar to [46]), and *Chameleon*, which uses input and output length, and adapter size.

Figure 19 shows the P99 TTFT latency for *OutputOnly* and *Chameleon* for the different output predictor accuracies as a function of time. We see that the system is robust to predictor accuracy for most of the time. However, during a load burst (at around 300s), the configurations with 60% accuracy have high TTFT latency. Also, the configuration that uses only the predicted output length (*OutputOnly*) is more sensitive to the predictor accuracy than *Chameleon*. Finally, with a predictor of 80% accuracy, *Chameleon* has approximately the same TTFT latency as with one of 100% accuracy.

2. Impact of the distribution of adapter rank popularity and adapter popularity within a rank. By default, our experiments use a uniform distribution for adapter rank popularity and a power-law distribution for adapter popularity within a rank (Section 5.1). In this section, we examine other distributions: i) uniform rank popularity and uniform adapter popularity within a rank *(U-U)*, ii) uniform rank and power-law adapter popularity *(U-P)*, and iii) power-law rank popularity and power-law adapter popularity *(P-P)*. Figure 20-right shows the normalized P99 TTFT latency with these distributions in *S-LoRA* and *Chameleon*. We see that both *S-LoRA* and *Chameleon* perform best under the *P-P* distribution, as the cost of loading adapters and the queuing delays decrease. *Chameleon*'s advanced caching and scheduling keep the P99 TTFT latency minimal for all distributions.

3. Impact of the total number of adapters. In our experiments, we have used a total number of adapters (N_a) equal to 100. In this section, we consider N_a equal to 10, 50, 100, 150, and 200 [49, 60].

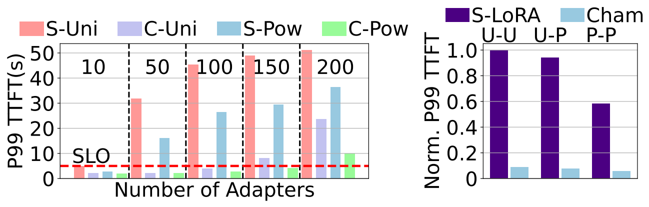

Figure 20: P99 TTFT latency sensitivity to the total number of adapters (left) and to their distribution (right).

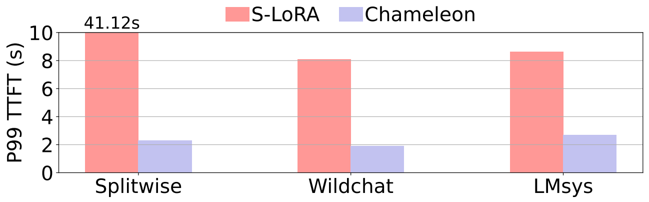

Figure 21: P99 TTFT latency for different traces. The SLOs for Splitwise, WildChat-1M, and LMSYS-Chat-1M are 5s, 3.3s, and 3.5s, respectively.

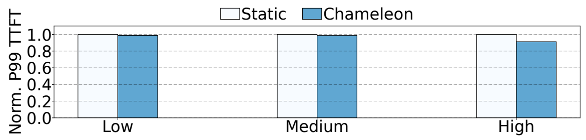

Figure 22: P99 TTFT latency for *Chameleon* normalized to a static scheme for different loads.

We also consider both uniform and power-law distributions for the rank popularity. Figure 20-left shows the P99 TTFT latency for *S-LoRA* (S) and *Chameleon* (C) for uniform (Uni) and power-law (Pow) distributions. The load is 9.5 RPS and the SLO is 5s. We see that *Chameleon* keeps the TTFT under SLO for up to 100 adapters when using a uniform distribution, and up to 150 when using a power-law distribution. In contrast, *S-LoRA* can only meet SLO for either distribution for 10 adapters. As the number of adapters increases, *Chameleon* keeps the TTFT latency low because: i) its adapter cache minimizes the increasing overheads of adapter loading and ii) its scheduler reduces the increasing effect of HoL blocking.

4. Impact of Additional Traces. We now use different traces beyond those from Splitwise [41] to evaluate *Chameleon*, without re-adjusting *Chameleon*'s tuned parameters—i.e., the coefficients in its cache eviction policy and WRS formula. We obtain traces from two data-sets: WildChat-1M [65] and LMSYS-Chat-1M [67]. In Figure 21, we plot the P99 TTFT latency for each trace for 9.5 RPS. The SLOs for Splitwise, WildChat-1M, and LMSYS-Chat-1M are 5s, 3.3s, and 3.5s, respectively. The new traces have generally smaller input and output lengths and thus their requests have shorter runtimes compared to Splitwise. In the figure, we see that *S-LoRA* fails to meet the SLO under high load for all traces due to queuing. In contrast, *Chameleon* meets the SLOs for all traces, and reduces the TTFT latency in the new traces by about 4× over *S-LoRA*.

5. Impact of the Scheduling Queue Organization. *Chameleon* uses K-means clustering to decide the number of scheduling queues and their cut-offs. It then uses the equations in Section 4.3.5 to assign resource quotas to queues. Further, it performs all these actions dynamically. In this section, we compare *Chameleon* to a static system that, knowing the smallest and the largest size of

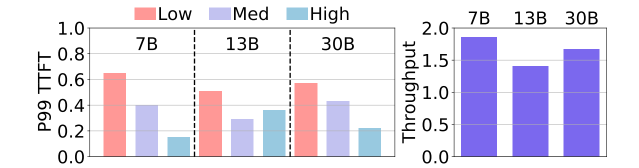

Figure 23: Normalized P99 TTFT latency (left) and throughput (right) of *Chameleon* over *S-LoRA* with different LLMs (Llama-7B, 13B, and 30B) and loads (Low, Medium, and High).

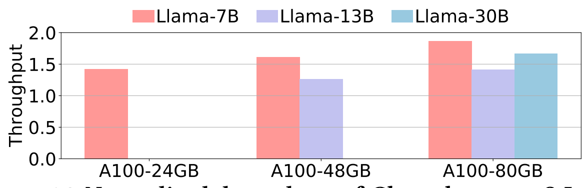

Figure 24: Normalized throughput of *Chameleon* over *S-LoRA* with different GPU memory sizes (24GB, 48GB, and 80GB) and LLMs sizes (Llama-7B, 13B, and 30B).

requests, sets the number of queues to 4, sets their ranges equally, and assigns the number of resource tokens to each queue equally. We call the system *Static*. Figure 22 shows the normalized P99 TTFT latency for *Static* and *Chameleon*. We see that, for low and medium load, the two configurations perform similarly. For high load, *Chameleon*'s design reduces the TTFT latency by 10%.

5.5 Scalability Analysis

To assess the scalability of Chameleon, we run experiments with larger models (Llama-7B, Llama-13B, and Llama-30B) and with different memory capacities (24GB, 48GB, and 80GB). In this section, we run the experiments on an A100 NVIDIA GPU that, by default, has 80GB of memory. Given the available memory space, we use 500, 100, and 10 different adapters in the experiments with 7B, 13B, and 30B parameter models, respectively.

1. Scalability with LLM size. In this section, we increase the size of the base LLM model and the load in the system. Figure 23-left shows the P99 TTFT latency of *Chameleon* for Llama-7B, 13B, and 30B, and for low, medium, and high loads. The latency for a given model and load is normalized to *S-LoRA*'s for the same model and load. We see that *Chameleon* always has a substantially lower TTFT latency than *S-LoRA*. Overall, averaged across all loads, *Chameleon* reduces the P99 TTFT latency over *S-LoRA* by 60.0%, 61.3%, and 59.3% for the Llama-7B, Llama-13B, and Llama-30B models, respectively.

Figure 23-right shows the throughout of *Chameleon* normalized to that of *S-LoRA* for different LLM sizes. We see that *Chameleon* improves the throughput by 1.86×, 1.41×, and 1.67× for the Llama-7B, Llama-13B, and Llama-30B models, respectively.

2. Scalability with GPU memory size. In this section, we increase the GPU memory size in an A100 GPU. Figure 24 shows the normalized throughput of *Chameleon* over *S-LoRA* as we increase the GPU memory size (24GB, 48GB, and 80GB) and the LLM size (Llama-7B, 13B, and 30B). Llama-30B fits only in 80GB of memory, Llama-13B fits in 48GB and 80GB of memory, and Llama-7B fits in all memory configurations. We see that *Chameleon* is more effective at increasing the throughput over *S-LoRA* as the amount of GPU memory increases. This is because a larger memory creates more space for adapter caching. For example, *Chameleon* improves the

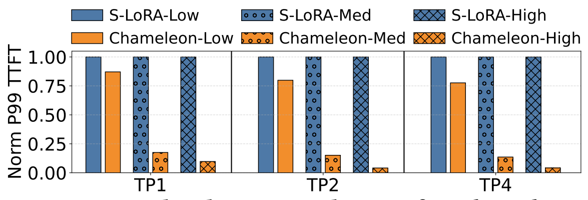

Figure 25: Normalized P99 TTFT latency for *Chameleon* and *S-LoRA* with different levels of tensor parallelism (TP1, TP2, and TP4), and request load (Low, Medium, and High).

throughput of Llama-7B over *S-LoRA* by 1.4×, 1.6×, and 1.9× with 24GB, 48GB, and 80GB of GPU memory, respectively.

3. Scalability with GPU compute capability. This section compares the throughput improvements of *Chameleon* running on different hardware platforms with the same memory capacity. We consider an A40 GPU with 48GB of memory and 100 adapters, and an A100 GPU with 48GB of memory and 500 adapters. The first platform is discussed in Section 5.2.2 and Figure 11. *Chameleon* is shown to improve the throughput over *S-LoRA* by 1.5×. The second platform is discussed in Section 5.5.2. As shown in the second bar of Figure 24, *Chameleon* improves the throughput over *S-LoRA* by 1.6×. Therefore, *Chameleon*'s improvement in throughput over *S-LoRA* increases with the more powerful GPU, even with more adapters.

5.6 Multi-GPU Experiments

Finally, we evaluate *Chameleon* in a multi-GPU environment. We use the A100 server with 4 GPUs and employ tensor parallelism (TP) with 2 or 4 GPUs. We examine Low, Medium, and High request loads. In this setup, the base LLaMA-7B model and the adapters are partitioned across the GPUs along tensor dimensions. *Chameleon*'s caching and scheduling mechanisms operate as in the single-GPU case. The cache is distributed across the GPUs, storing partitions of adapters, while scheduling continues to treat all GPUs as a single execution engine. No changes are made to the caching or scheduling policies to accommodate the multi-GPU setup.

Figure 25 compares the P99 TTFT latencies of *Chameleon* and *S-LoRA* for TP1, TP2, and TP4, and different request loads. The bars are normalized to *S-LoRA* for the specific level of parallelism and load. We see that *Chameleon* reduces the TTFT latency across all parallelism and load levels. The reduction widens with increasing parallelism. This is because, with more GPUs, the cost of loading adapters onto all participating GPUs becomes a bigger bottleneck in *S-LoRA*. *Chameleon*'s ability to cache and reuse adapter fragments across GPUs helps it avoid this overhead and scale more efficiently. This effect gets accentuated at higher loads. Overall, the gains of *Chameleon* are substantial: for TP4 and High load, *Chameleon* reduces the P99 TTFT latency by 95.8% over *S-LoRA*.

6 Related Work

LLM Inference Optimizations. Many works proposed hardware [6, 16, 19, 22, 23, 41, 44, 45, 62–64, 69], algorithm [9, 14, 18] and system-level [1, 20, 31, 32, 61] optimizations for performance and energy-efficiency [40, 52, 53] of LLM inference systems. These works consider LLMs with only a base model and do not optimize for a multi-adapter LLM inference environment. Chameleon is mostly orthogonal to such techniques and can be combined with them.

LLM Inference with Parameter-Efficient Fine Tuning. Since the adoption of parameter-efficient fine tuning techniques [17, 24, 27, 58], researchers have been working on optimizing the system stack for efficient LLM inference in multi-adapter environments [4, 25, 49, 60, 68]. S-LoRA [49] and Punica [4] decouple the base model from task-specific adapters and fetch the required adapters on the fly from the host to the GPU memory. dLoRA [60] dynamically merges and unmerges adapters with the base model based on the current system state. In the paper, we quantitatively compare Chameleon to S-LoRA as the state-of-the-art baseline.

LLM Inference Scheduling. Many works explored scheduling policies for LLM inference serving [11, 39, 46, 50, 54, 59, 61]. μServe [46] and Learning to Rank [11] reduce the HoL blocking effects via SJF scheduling. We quantitatively compare to μServe. Based on input and output request lengths, ExeGPT [39] and DynamoLLM [53] allocate resources (batch size and model parallelism) and schedule the requests for minimal cost and energy consumption, respectively. Llumnix [54] reschedules the requests across worker replicas to improve load balance. These works focus on scheduling LLM inference requests in a multi-node environment, while using conventional iteration-level scheduling [61] within a node. Chameleon redesigns the scheduling policy within a node, and can be combined with cluster-level schedulers.

General-Purpose Workload Scheduling. Size-Interval Task Assignment (SITA) [7, 15] addresses head-of-line blocking by providing an "express-lane" for short tasks. Q-Zilla [35] leverages this idea and proposes a Server-Queue Decoupled Size-Interval Task Assignment for highly diverse microservice invocations. Chameleon applies the algorithm to a new domain: multi-adapter LLM inference serving. Moreover, SITA assumes perfect knowledge of task size, while Q-Zilla relies on request preemption. On the other hand, Chameleon uses a predictor for a request's output length (which is unknown ahead of time), and does not use preemption due to its high cost in LLM inference environments [20, 46, 59].

7 Conclusion

This paper presented Chameleon, an efficient LLM inference serving system for many-adapter environments. Chameleon introduces two new ideas: adapter caching and adapter-aware request scheduling. Caching minimizes the overhead of loading the adapter weights on the request's critical path, while scheduling alleviates head-of-line blocking and starvation for requests with highly-diverse execution times. Under high loads, Chameleon reduces the P99 TTFT latency by 80.7% and the P50 TTFT latency by 48.1% over a state-of-the-art-baseline, while improving the throughput by 1.5×.

Acknowledgments

This work was supported by NSF under grants CCF 2107470 and CCF 2316233; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by the IBM-Illinois Discovery Accelerator Institute; and by an Amazon Fellowship funded by the UIUC AICE Center.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI*

24). USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. arXiv:2312.11514 [cs.CL]

[3] Jairus Bowne. 2024. Using Large Language Models in Learning and Teaching. https://biomedicalsciences.unimelb.edu.au/study/dlh/assets/documents/large-language-models-in-education/llms-in-education.

[4] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024. Punica: Multi-Tenant LoRA Serving. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 1–13. https://proceedings.mlsys.org/paper_files/paper/2024/file/054de805fcceb78a201f5e9d53c85908-Paper-Conference.pdf

[5] Ludmila Cherkasova. 1998. *Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy.* Technical Report 98-69 (R.1). HP Labs.

[6] Jaewan Choi, Jaehyun Park, Kwanhee Kyung, Nam Sung Kim, and Jung Ho Ahn. 2024. Unleashing the Potential of PIM: Accelerating Large Batched Inference of Transformer-Based Generative Models. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '24)*.

[7] Mark E. Crovella, Mor Harchol-Balter, and Cristina D. Murta. 1998. Task assignment in a distributed system (extended abstract): improving performance by unbalancing load. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (Madison, Wisconsin, USA) *(SIGMETRICS '98/PERFORMANCE '98)*. Association for Computing Machinery, New York, NY, USA, 268–269. https://doi.org/10.1145/277851.277942

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG]

[9] Jyotikrishna Dass, Shang Wu, Huihong Shi, Chaojian Li, Zhifan Ye, Zhongfeng Wang, and Yingyan Lin. 2023. ViTALiTy: Unifying Low-rank and Sparse Approximation for Vision Transformer Acceleration with a Linear Taylor Attention. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '23)*.

[10] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

[11] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient LLM Scheduling by Learning to Rank. arXiv:2408.15792 [cs.LG] https://arxiv.org/abs/2408.15792

[12] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. New York, NY, USA. https://doi.org/10.1145/3445814.3446757

[13] GitHub. 2024. The world's most widely adopted AI developer tool. https://github.com/features/copilot.

[14] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*.

[15] Mor Harchol-Balter, Mark E. Crovella, and Cristina D. Murta. 1999. On Choosing a Task Assignment Policy for a Distributed Server System. *J. Parallel and Distrib. Comput.* 59, 2 (1999), 204–228. https://doi.org/10.1006/jpdc.1999.1577

[16] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[17] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2106.09685

[18] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. 2024. Pre-gated MoE: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*.

[19] Hongsun Jang, Jaeyong Song, Jaewon Jung, Jaeyoung Park, Youngsok Kim, and Jinho Lee. 2024. Smart-Infinity: Fast Large Language Model Training using Near-Storage Processing on a Real System. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '24)*.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with Paged Attention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[21] Marina Lammertyn. 2024. 60+ ChatGPT Statistics And Facts You Need to Know in 2024. https://blog.invgate.com/chatgpt-statistics.

[22] Jungi Lee, Wonbeom Lee, and Jaewoong Sim. 2024. Tender: Accelerating Large Language Models via Tensor Decomposition and Runtime Requantization. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*.

[23] SeungYul Lee, Hyunseung Lee, Jihoon Hong, SangLyul Cho, and Jae W. Lee. 2024. VGA: Hardware Accelerator for Scalable Long Sequence Model Inference. In *Proceedings of the International Symposium on Microarchitecture (MICRO '24)*.

[24] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. arXiv:2104.08691 [cs.CL] https://arxiv.org/abs/2104.08691

[25] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. 2024. CaraServe: CPU-Assisted and Rank-Aware LoRA Serving for Generative LLM Inference. arXiv:2401.11240 [cs.DC] https://arxiv.org/abs/2401.11240

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*.

[27] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2022. P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks. arXiv:2110.07602 [cs.CL] https://arxiv.org/abs/2110.07602

[28] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning Methods. https://github.com/huggingface/peft.

[29] Meta. 2024. Llama3-70B. https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct.

[30] Meta AI. 2024. Open Pre-trained Transformer Language Models. https://huggingface.co/docs/transformers/model_doc/opt.

[31] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[32] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[33] Microsoft Copilot. 2025. Microsoft Copilot — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Microsoft_Copilot [Online; accessed 22-August-2025].

[34] Amirhossein Mirhosseini and Thomas Wenisch. 2021. μSteal: a theory-backed framework for preemptive work and resource stealing in mixed-criticality microservices. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 102–114. https://doi.org/10.1145/3447818.3463529

[35] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. 2020. Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 207–219. https://doi.org/10.1109/HPCA47549.2020.00026

[36] Mistral AI. 2024. The Mixtral-8x22B Large Language Model. https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1.

[37] NVIDIA. 2024. Introduction to the NVIDIA DGX A100 System. https://docs.nvidia.com/dgx/dgxa100-user-guide/introduction-to-dgxa100.html.

[38] NVIDIA. 2024. NVIDIA A40 Data Center GPU. https://www.nvidia.com/en-us/data-center/a40/.

[39] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. 2024. Characterizing Power Management Opportunities for LLMs in the Cloud. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[41] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*.

[42] Cheng Peng, Xi Yang, Aokun Chen, Kaleb Smith, Nima PourNejatian, Anthony Costa, Cheryl Martin, Mona Flores, Ying Zhang, Tanja Magoc, Gloria Lipori, Mitchell Duane, Naykky Ospina, Mustafa Ahmed, William Hogan, Elizabeth Shenkman, Yi Guo, Jiang Bian, and Yonghui Wu. 2023. A study of generative large language model for medical research and healthcare. *npj Digital Medicine* (2023).

[43] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterHub: A Framework for Adapting Transformers. *arXiv preprint arXiv:2007.07779* (2020). https://arxiv.org/abs/2007.07779

[44] Yubin Qin, Yang Wang, Dazheng Deng, Zhiren Zhao, Xiaolong Yang, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. 2023. FACT: FFN-Attention Co-optimized Transformer Architecture with Eager Correlation Prediction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23).*

[45] Yubin Qin, Yang Wang, Zhiren Zhao, Xiaolong Yang, Yang Zhou, Shaojun Wei, Yang Hu, and Shouyi Yin. 2024. MECLA: Memory-Compute-Efficient LLM Accelerator with Scaling Sub-matrix Partition. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24).*

[46] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Power-aware Deep Learning Model Serving with μ-Serve. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 75–93. https://www.usenix.org/conference/atc24/presentation/qiu

[47] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[48] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20).*

[49] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph Gonzalez, and Ion Stoica. 2024. S-LoRA: Scalable Serving of Thousands of LoRA Adapters. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 296–311. https://proceedings.mlsys.org/paper_files/paper/2024/file/906419cd502575b617cc489a1a696a67-Paper-Conference.pdf

[50] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 965–988. https://www.usenix.org/conference/osdi24/presentation/sheng

[51] Jovan Stojkovic, Esha Choukse, Chaojie Zhang, Inigo Goiri, and Josep Torrellas. 2024. Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference. *arXiv e-prints*, Article arXiv:2403.20306 (March 2024), arXiv:2403.20306 pages. https://doi.org/10.48550/arXiv.2403.20306 arXiv:2403.20306 [cs.AI]

[52] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Esha Choukse, Haoran Qiu, Rodrigo Fonseca, Josep Torrellas, and Ricardo Bianchini. 2025. *TAPAS: Thermal- and Power-Aware Scheduling for LLM Inference in Cloud Platforms.* Association for Computing Machinery, New York, NY, USA, 1266–1281. https://doi.org/10.1145/3676641.3716025

[53] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1348–1362. https://doi.org/10.1109/HPCA61900.2025.00102

[54] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24).*

[55] Technology Innovation Institute (TII). 2024. Falcon-180B. https://huggingface.co/tiiuae/falcon-180B. (2024).

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

[57] Tanay Varshney. 2024. Build an LLM-Powered Data Agent for Data Analysis. https://developer.nvidia.com/blog/build-an-llm-powered-data-agent-for-data-analysis/.

[58] Zhengbo Wang, Jian Liang, Ran He, Zilei Wang, and Tieniu Tan. 2025. LoRA-Pro: Are Low-Rank Adapters Properly Optimized?. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=gTwRMU3lJ5

[59] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. arXiv:2305.05920 [cs.LG]

[60] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 911–927. https://www.usenix.org/conference/osdi24/presentation/wu-bingyang

[61] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[62] Zhongkai Yu, Shengwen Liang, Tianyun Ma, Yunke Cai, Ziyuan Nan, Di Huang, Xinkai Song, Yifan Hao, Jie Zhang, Tian Zhi, Yongwei Zhao, Zidong Du, Xing Hu, Qi Guo, and Tianshi Chen. 2024. FlashLLM: A Chiplet-Based In-Flash Computing Architecture to Enable On-Device Inference of 70B LLM. In *Proceedings of the International Symposium on Microarchitecture (MICRO '24).*

[63] Sungmin Yun, Kwanhee Kyung, Juhwan Cho, Jaewan Choi, Jongmin Kim, Byeongho Kim, Sukhan Lee, Kyomin Sohn, and Jung Ho Ahn. 2024. Duplex: A Device for Large Language Models with Mixture of Experts, Grouped Query Attention, and Continuous Batching. In *Proceedings of the International Symposium on Microarchitecture (MICRO '24).*

[64] Hengrui Zhang, August Ning, Rohan Baskar Prabhakar, and David Wentzlaff. 2024. LLMCompass: Enabling Efficient Hardware Design for Large Language Model Inference. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24).*

[65] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. WildChat: 1M ChatGPT Interaction Logs in the Wild. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=Bl8u7ZRlbM

[66] Youpeng Zhao Zhao, Di Wu Wu, and Jun Wang. 2024. ALISA: Accelerating Large Language Model Inference via Sparsity-Aware KV Caching. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24).*

[67] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs.CL] https://arxiv.org/abs/2309.11998

[68] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. 2022. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '22).*

[69] Hanqing Zhu, Jiaqi Gu, Hanrui Wang, Zixuan Jiang, Zhekai Zhang, Rongxing Tang, Chenghao Feng, Song Han, Ray T. Chen, and David Z. Pan. 2024. Lightening-Transformer: A Dynamically-Operated Optically-Interconnected Photonic Transformer Accelerator. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '24).*