Data-Driven Optimization of GPU efficiency for Distributed LLM–Adapter Serving

Ferran Agulló[a,b,1,*], Joan Oliveras[a,b], Chen Wang[c], Alberto Gutierrez-Torre[a], Olivier Tardieu[c], Alaa Youssef[c], Jordi Torres[a,b], Josep Ll. Berral[a,b]

[a] *Barcelona Supercomputing Center (BSC), Barcelona, Spain*
[b] *Universitat Politècnica de Catalunya - BarcelonaTech (UPC), Barcelona, Spain*
[c] *IBM Research, New York, USA*

Abstract

Large Language Model (LLM) adapters enable low-cost model specialization, but introduce complex caching and scheduling challenges in distributed serving systems where hundreds of adapters must be hosted concurrently. While prior work has largely focused on latency and throughput optimization, minimizing GPU resource requirements through near-peak utilization remains largely underexplored. This paper presents a data-driven pipeline that, for a given workload, computes an adapter placement that serves the workload with the minimum number of GPUs while avoiding request starvation and GPU memory errors. To that end, the approach identifies the maximum feasible throughput attainable on each GPU by leveraging accurate performance predictions learned from real serving behavior. The proposed pipeline integrates three components: (i) a Digital Twin (DT) tailored to LLM-adapter serving, (ii) a distilled machine learning (ML) model trained on DT-generated data, and (iii) a greedy placement algorithm that exploits ML-based performance estimates to maximize GPU efficiency. The DT emulates real system dynamics with high fidelity, achieving below 5% throughput estimation error while executing up to 90× faster than full LLM benchmarking across both predictable and unpredictable workloads. The learned ML models further accelerate performance estimation with marginal accuracy degradation, enabling scalable optimization. Experimental results demonstrate that the pipeline substantially improves GPU efficiency, reducing the number of GPUs required to sustain target workloads by 60% on average across the evaluated scenarios. Beyond GPU efficiency, the pipeline can be adapted to alternative objectives, such as latency minimization, highlighting its versatility for future large-scale LLM serving infrastructures.

Keywords: Large language models (LLMs), Distributed system, LLM adapters, LoRA adapters, Digital twin, GPU optimization, Machine learning (ML), Performance modeling

1. Introduction

With the rapid advancement and widespread adoption of Large Language Models (LLMs), the demand for LLM-adapters has grown significantly. While LLMs are large-scale models trained to achieve strong performance across diverse language tasks, adapters specialize this general knowledge to concrete applications through lightweight parameter additions [12, 11, 20, 10]. The use of adapters is far quicker than training-from-scratch or fine-tuning a new model, which is a tedious and long process that requires meticulous data curation and high computation capabilities. Moreover, adapters achieve comparable performance to other equivalent methods such as in-context learning [4] or prompt tuning [19, 25].

As adapters are increasingly used to specialize and deploy LLMs across diverse tasks [35], efficiently managing their inference execution has become a key systems concern [37, 18, 13]. Although adapters were originally designed to be merged into the backbone model to avoid additional computation and preserve latency [12], contemporary serving systems typically keep them unmerged. This design enables the serving of multiple task-specific adapters on top of a shared backbone without replicating its large parameter footprint in GPU memory. Such an approach is particularly well suited for multi-tenant environments, where a system must serve multiple users, each requiring a distinct model specialization [30]. Since adapters are compact, a single GPU can serve hundreds or even thousands of different specializations of the same LLM [31, 5]. This consolidation increases per-GPU throughput by aggregating requests from the hosted adapters whose individual arrival rates are often too low to saturate a GPU.

However, excessive adapter packing can cross a critical threshold where *request starvation* emerges. In this regime, the GPU lacks sufficient memory for intermedi-

*Corresponding author
Email addresses: ferran.agullo@bsc.es (Ferran Agulló), joan.oliveras@bsc.es (Joan Oliveras), chen.wang1@ibm.com (Chen Wang), alberto.gutierrez@bsc.es (Alberto Gutierrez-Torre), tardieu@ibm.com (Olivier Tardieu), asyousse@ibm.com (Alaa Youssef), jordi.torres@bsc.es (Jordi Torres), josep.ll.berral@upc.edu (Josep Ll. Berral)

[1] Code will be available at: https://github.com/FerranAgulloLopez/DistributedEfficientAdapterLLMServing

ate request states (i.e. KV cache), so requests accumulate faster than they are processed and latency increases steadily. Determining an allocation that maximizes per-GPU throughput, and thus GPU utilization, without triggering starvation is non-trivial. This optimal point, denoted as Max_{pack}, depends on the interplay between the sizes and request arrival rates of the adapters, which vary across deployments and over time. Adapter size constrains the GPU memory available for intermediate states and increases computation time, while arrival rate governs the load induced by each adapter.

To mitigate the GPU footprint of adapter weights, serving systems often employ dynamic mechanisms that swap adapters between CPU and GPU memory based on utilization. Adapters resident in GPU memory can execute requests in parallel with other resident adapters [6], while swapping enables interleaved execution among adapters that do not simultaneously coexist on the GPU. Some frameworks, such as vLLM [16], additionally enforce a static upper bound on the number of loaded adapters, A_{max}, and preallocate the corresponding GPU memory at initialization. This choice directly shapes Max_{pack}: increasing A_{max} reduces memory available for request processing, while decreasing it limits achievable parallelism. Improper tuning of A_{max} can therefore induce starvation and, in extreme cases, GPU memory errors.

Building on these observations, this work addresses the following problem: **given an expected future workload composed of adapters with heterogeneous sizes and request arrival rates, determine a GPU allocation strategy that maximizes per-GPU throughput by achieving Max_{pack}, so that the workload is served using the minimum number of GPUs without incurring starvation or memory errors**. The solution must also derive the corresponding A_{max} configuration for each GPU. By *expected future workload*, we focus on *predictable* workloads that can be estimated in advance, such as long-term patterns present in production traces that exhibit periodicity [37]. This formulation enables distributed systems to provision the minimal GPU capacity required for a workload in advance, improving hardware efficiency. We refer to this optimization challenge as the **adapter caching problem**, whose output specifies adapter placement across GPUs together with per-GPU A_{max} values.

While prior work has explored the optimization of LLM-adapter serving via kernel-level enhancements [6], memory management techniques [31], and scheduling strategies [13], the adapter caching problem remains largely underexplored. The most closely related approaches, such as dLoRA [37] and LoRAServe [14], propose proactive adapter placement strategies based on estimated long-term workload patterns, aiming to reduce latency or improve throughput by fully leveraging available hardware resources. In contrast, our objective is to maximize hardware efficiency by minimizing the number of required GPUs through near-peak utilization of a subset of the devices, while leaving the remaining GPUs available for alternative workloads or reduced energy consumption.

To this end, we propose a data-driven pipeline comprising three phases: (i) a *Digital Twin* (DT) that emulates an online LLM-adapter serving system; (ii) a machine learning (ML) phase that employs data generated by the DT; and (iii) a greedy algorithm that computes the final adapter placement to solve the caching problem. Unlike prior approaches, which rely on heuristics or partial knowledge of serving behavior, our method bases allocation decisions on complete serving behavior that is learned by the ML phase under diverse workload and system conditions. The greedy algorithm exploits this performance knowledge to approach the optimal packing point, Max_{pack}, while determining an appropriate A_{max} configuration. To train the ML models without incurring the prohibitive cost of profiling a real LLM-adapter serving system, we introduce the DT, which reproduces system behavior with orders-of-magnitude faster execution and substantially lower resource consumption. Building this DT required an in-depth profiling and analysis of the dominant overheads in LLM-adapter serving, which we also report.

We evaluate our approach using the widely adopted vLLM framework [16] with LoRA adapters [12]. To illustrate the generality of the adapter caching problem beyond a single framework, we additionally provide a brief analysis using S-LoRA [31] in Appendix A.

In summary, we make the following contributions:

- We propose a **data-driven pipeline for addressing the *adapter caching problem***, which aims to minimize the number of GPUs required to serve an anticipated workload. The approach maximizes per-GPU utilization while preventing request starvation and memory errors. The pipeline integrates a Digital Twin, an ML learning phase, and a greedy allocation algorithm. Results demonstrate that the proposed solution improves resource efficiency, reducing the number of GPUs required to sustain target workloads by an average of 60% across evaluated scenarios.

- To the best of our knowledge, we introduce **the first Digital Twin for LLM-adapter serving**. The proposed Digital Twin operates orders of magnitude faster than the real system while accurately reproducing key performance metrics, and enables efficient generation of synthetic data to support the ML phase.

- We provide a **comprehensive analysis of the dominant overheads in LLM-adapter serving**, quantifying their interactions across diverse workloads and deriving actionable guidelines for system configuration and optimization.

Paper structure: The remainder of this work is organized as follows. Sections 2 and 3 present the necessary background and prior work, with the former also showing the adapter caching problem in practice. Sections 4, 5, 6,

and 7 detail the components of the proposed pipeline. Section 8 reports the main experimental results, while Sections 9 and 10 provide discussion and concluding remarks.

2. Background

2.1. LLM serving

An LLM serving system processes each request through two sequential phases: *prefill* and *decode*. During prefill, all input tokens are processed in parallel, producing intermediate attention states, commonly referred to as the KV cache, which are stored in GPU memory to avoid redundant computation during generation. In the decode phase, output tokens are generated autoregressively, while newly computed KV values are appended to memory. Generation terminates upon emission of an end-of-sequence token or when the maximum output length is reached.

Because decoding is inherently sequential and therefore not compute-intensive, serving systems improve hardware utilization by processing multiple requests in parallel through batching [27]. As static batching is inefficient under variable output lengths, modern systems instead adopt *continuous batching* [40, 16, 23, 22]. It allows requests to dynamically enter and exit the batch between decoding iterations, significantly improving throughput and latency.

At runtime, the number of active requests in the batch is primarily constrained by GPU memory, as each request maintains a growing KV cache throughout decoding. Early frameworks conservatively preallocated memory for the maximum possible output length, resulting in substantial overprovisioning. In contrast, systems such as vLLM [16] and S-LoRA [31] employ greedy KV-cache allocation strategies that reserve memory only for a limited window of upcoming tokens, improving memory efficiency.

2.2. (vLLM) Adapter serving

Multiple types of LLM-adapters have been proposed [11, 20, 10, 32]. In this work, we focus on LoRA adapters [12], which remain the most widely adopted approach. LoRA introduces trainable weights into specific layers of the backbone LLM through two low-rank matrices. Their resulting activations are added to the corresponding backbone layer outputs, allowing the adapter to learn only the residual difference between the pretrained and target tasks. The size of a LoRA adapter is defined by the dimensionality of its low-rank latent space, known as *rank*.

Modern serving systems support parallel execution of multiple adapters within the same batch through kernel-level optimizations [6]. Nevertheless, a request can be processed only if both the backbone model and adapter weights reside in GPU memory. Adapter weights therefore reduce the memory available for requests KV cache. To alleviate this limitation, adapters are dynamically swap between CPU and GPU depending on their current usage.

In vLLM, GPU memory is statically partitioned at initialization by reserving a fixed region for adapter weights.

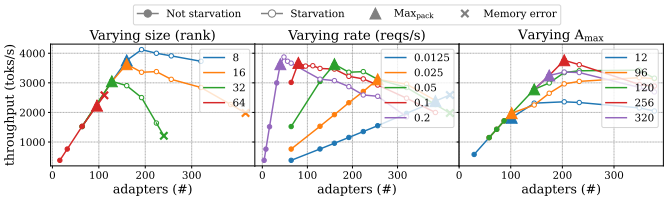


Figure 1: Throughput as a function of the number of served adapters under varying adapter sizes (left), arrival rates (center) and configured A_{max} (right). Each line corresponds to experiments in which all parameters remain fixed except for the number of adapters. Results were obtained using vLLM with Llama-2-7B [33] and a public LoRA adapter [39] on an NVIDIA H100 Hopper GPU, processing a total of 4096 prompts submitted at the beginning of execution until completion. Default parameters were chosen to saturate the GPU between 100 and 300 adapters for improved visualization: adapter size 8, per-adapter arrival rate of 0.05 req/s, 250 input tokens per request and 231 output tokens per request. In the two leftmost plots, A_{max} is set equal to the number of served adapters and S_{max} is configured to match the adapter size used in each experiment across all plots, representing the most straightforward GPU configuration.

This design limits the maximum number of simultaneously loaded adapters, denoted as A_{max}. Although tunable, there is no principled methodology to choose its optimal value. Furthermore, vLLM assigns a uniform maximum memory footprint per adapter, S_{max}, causing all adapters to occupy identical GPU memory space regardless of their actual size. S-LoRA [31] mitigates these limitations by jointly managing KV-cache and adapter-weight memory within a unified cache. However, given the archival status of its codebase and its limited adoption in practice, it is not used as the primary framework in this study. An exploratory analysis is provided in Appendix A.

2.3. Illustrating the adapter caching problem

Fig. 1 illustrates the adapter caching problem under homogeneous workloads on a single GPU. Each curve shows the throughput as a function of the number of concurrently served adapters. Two distinct regimes are consistently observed in each line. Initially, throughput increases approximately linearly as additional adapters introduce more request load that can be efficiently batched together. Beyond a certain point, throughput saturates or degrades due to insufficient GPU memory for KV-cache allocation, resulting in request starvation. The transition between these two regimes marks the optimal packing point Max_{pack} targeted in this work, which maximizes throughput while avoiding starvation. In practice, we identify this point as the highest measured throughput that remains above 90% of the total incoming token rate.

The location of Max_{pack} is highly sensitive to adapter size, arrival rate, and the configured value of A_{max}, as a consequence of the adapter-serving overheads analyzed in Section 5.1. Larger adapters reduce the available memory for request processing and increase computation cost, yielding lower Max_{pack} points. Lower arrival rates similarly obtain lower peak points, as more adapters are required to saturate GPU memory, increasing adapter

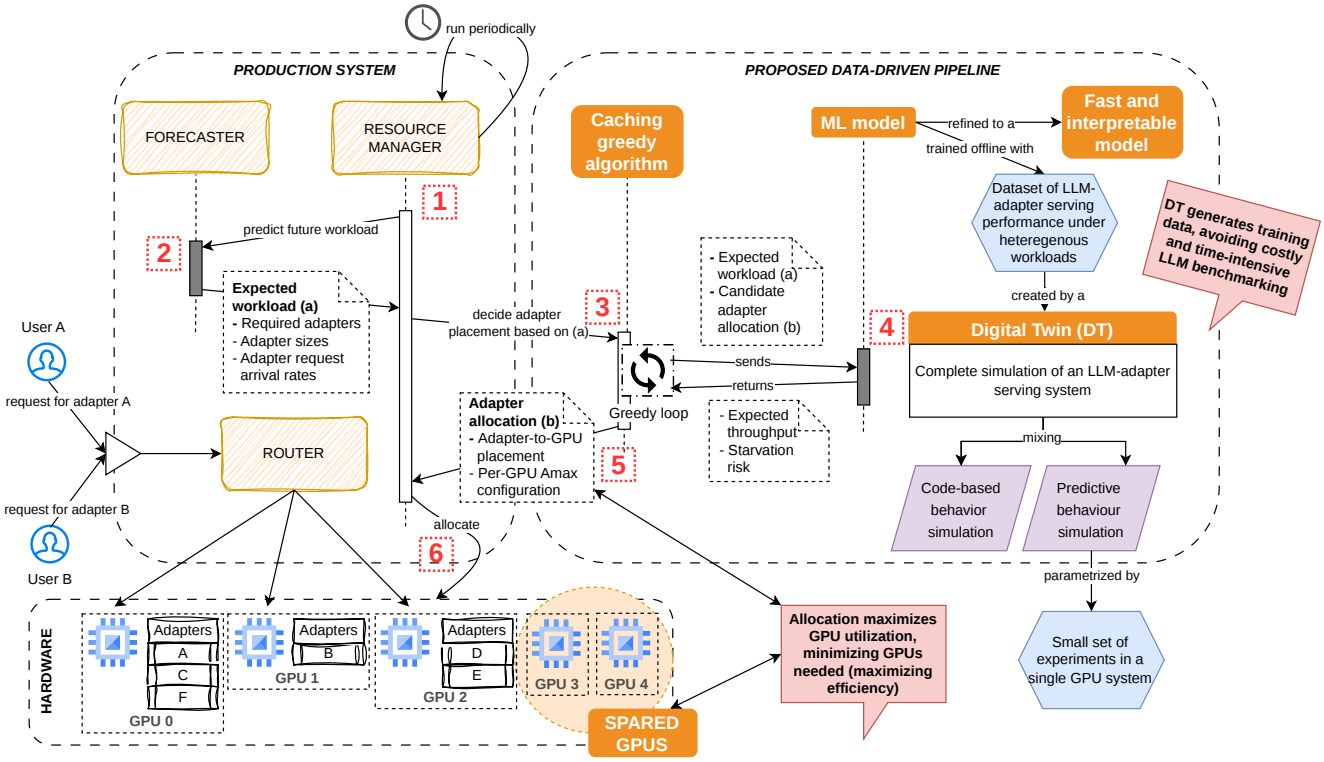


Figure 2: Proposed data-driven pipeline to address the adapter caching problem (right), shown alongside its expected usage within a production system (left). The numbered red markers indicate the recommended reading order of the workflow.

overheads. The choice of A_{max} introduces a trade-off between reserving excessive memory for adapters (e.g., $A_{max} = 320$) and restricting achievable parallelism (e.g., $A_{max} = 96$). While the results shown correspond to homogeneous workloads, higher variance arises under heterogeneous adapter sizes and arrival rates. Lastly, memory errors (marked as crosses) occur for large adapter sizes and low arrival rates as A_{max} is trivially set equal to the number of served adapters, resulting in the reservation of more GPU memory for adapter weights than is available.

3. Related work

3.1. LLM adapter optimization

A growing body of work has focused on improving the efficiency of LLM-adapter serving systems. Existing approaches can be broadly divided into three categories. First, kernel-level optimizations, such as Punica [6], improve execution efficiency by enabling batching across requests associated with different adapters. Second, memory-level optimizations, exemplified by S-LoRA [31], extend vLLM by dynamically partitioning GPU memory across both requests and adapters. Third, scheduling techniques, which include methods for request scheduling, such as Toppings [18] and Chameleon [13], as well as adapter placement strategies, discussed in the following section.

Toppings mitigates cold-start overheads and optimizes scheduling for heterogeneous adapter sizes, whereas Chameleon improves adapter cache management and introduces an express-lane mechanism for short requests. Both systems provide detailed performance analyses: Toppings primarily studies the computational overhead introduced by mixing adapters, while Chameleon focuses on Time To First Token (TTFT) and adapter loading latency. We extend these performance analyses by systematically studying the primary overheads of adapters and their impact on throughput and Inter-Token Latency (ITL), which form the foundation for the design of our proposed DT.

3.1.1. Placement methods

dLoRA [37] and LoRAServe [14] are prominent approaches to adapter placement. Both propose proactive placement strategies for distributed systems, analogous to the adapter caching problem, leveraging estimated long-term workload patterns derived from production traces to minimize latency and maximize throughput. dLoRA uses a heuristic based on the ratio between available GPU memory after loading adapter weights and the estimated adapter load, whereas LoRAServe uses a greedy algorithm that reduces adapter size heterogeneity by leveraging profiled single-adapter serving throughput as an estimate of peak GPU capacity. Additionally, dLoRA incorporates a reactive Integer Linear Programming (ILP) based optimization to adapt placement decisions to short-term workload variations arising from unpredictable output lengths.

While dLoRA and LoRAServe aim to fully utilize avail-

able hardware resources, our work focuses on resource efficiency by achieving near-peak GPU utilization (Max_{pack}), thereby reducing the total number of GPUs required to serve a given workload. Furthermore, unlike the heuristic-based approach of dLoRA, we adopt a data-driven methodology grounded in observed serving behavior through the DT. Compared to LoRAServe, our approach captures the dynamics of serving adapters under diverse workloads and A_{max} configurations, thereby reflecting the full complexity of multi-adapter LLM serving.

3.2. LLM simulators or digital twins

Several simulators have been proposed to model default LLM serving behavior. Vidur [1] aims to simplify server configuration selection by training random forest models on benchmarking data to predict performance metrics. LLMServingSim [7] emulates the continuous batching loop of LLM serving systems and is primarily intended for architecture exploration. Beyond simulators, more specific analytical performance models have also been proposed. Latency is commonly approximated as a linear function of batch size, following earlier studies on generic AI models [41, 29]. In the context of adapter serving, Toppings [18] models adapter computation latency as a linear function of batch size and either the sum or the maximum adapter size within the current batch.

To the best of our knowledge, no existing simulator or digital twin explicitly models the combined dynamics of adapter caching, KV-cache allocation, and continuous batching in multi-adapter LLM serving. Our Digital Twin combines code-based simulation with predictive behavior modeling. The latter relies on performance models that reuse existing analytical formulations where applicable, while incorporating additional components and refinements to accurately capture adapter-serving behavior under heterogeneous workloads.

3.3. Extension of prior workshop version

A preliminary version of this work was presented at the NeurIPS ML for Systems Workshop (2025) [21], where throughput maximization was addressed in single-GPU settings via Max_{pack} estimation. This manuscript substantially extends that work by generalizing the problem to distributed multi-GPU environments and introducing a greedy placement algorithm for adapter allocation. Moreover, the ML component is reformulated as a distilled surrogate of the Digital Twin (DT), enabling efficient performance prediction to guide placement under heterogeneous workloads. Finally, the experimental evaluation is significantly expanded, including validation under unpredictable arrival patterns and distributed scenarios with real trace-sampled arrivals not considered in the workshop version.

4. Pipeline overview

Fig. 2 illustrates the proposed data-driven pipeline and its expected integration within a production system. Inspired by the long-term proactive strategy of dLoRA, the pipeline is designed to be periodically reinvoked based on an anticipated workload. A workload is characterized by the required adapters, their sizes, and their request arrival rates. Given this workload, the pipeline computes an adapter-to-GPU placement together with the optimal A_{max} configuration per device. The resulting allocation maximizes GPU efficiency by driving a subset of devices to their maximum feasible packing throughput (Max_{pack}), thereby minimizing the number of GPUs required to serve the workload. The remaining devices can be reassigned to other workloads to improve overall system efficiency or powered down to reduce energy consumption.

The pipeline consists of three components: (i) a Digital Twin (DT), (ii) an ML learning phase, and (iii) a greedy adapter caching algorithm. The greedy algorithm produces the final placement decision, relying on performance predictions generated by the ML models. Trained offline, these models estimate the achievable throughput of a GPU under a given adapter placement and A_{max} configuration, and predict whether starvation may arise. An optional refinement phase can simplify the models into faster and more interpretable surrogates, at the cost of limited accuracy degradation.

Training such models requires a large and diverse dataset capturing LLM-adapter serving behavior across heterogeneous workloads. Exhaustively constructing this dataset via real-system benchmarking is computationally prohibitive due to its execution time and resource cost. To overcome this limitation, we introduce a Digital Twin that emulates the internal dynamics of an LLM-adapter serving system through a combination of code-based and predictive behavior simulation. The DT operates orders of magnitude faster and with substantially lower resource consumption than full-system benchmarking, enabling large-scale synthetic data generation to train the ML models. While the DT provides high-fidelity performance estimates, it is not directly invoked by the greedy caching algorithm, as the ML surrogate enables significantly faster inference and improved interpretability, making the pipeline suitable for production deployment. Prior to use, the DT requires a lightweight parameterization phase based on a small set of benchmarking experiments executed on the target hardware and model configuration.

5. Digital twin

The proposed Digital Twin (DT) reproduces an online LLM–adapter serving system. Rather than acting as a simple performance metric estimator, the DT implements the code loop corresponding to the continuous batching process characteristic of modern serving frameworks. This design enables accurate performance estimation under heterogeneous workload distributions. The DT operates offline and exclusively on CPU to facilitate cost-efficient generation of training data for the ML learning phase.

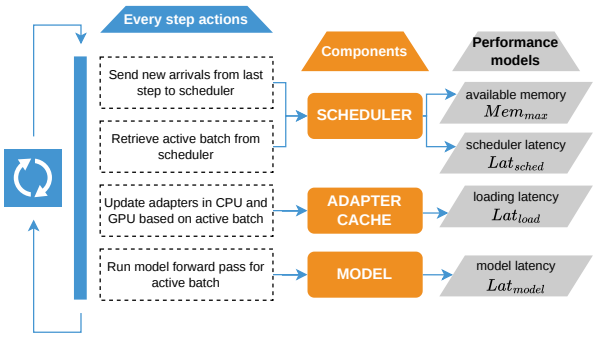


Figure 3: Digital Twin behavior and architecture.

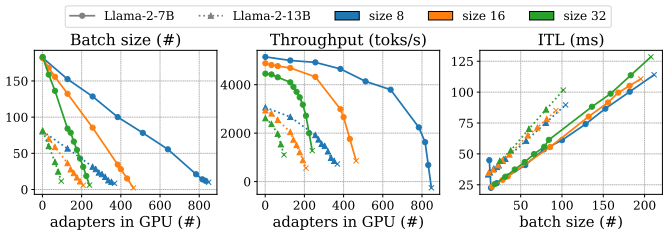


Figure 4: Evolution of batch size and throughput with increasing numbers of loaded adapters (left, center) and ITL versus batch size (right), across models and adapter sizes. Crosses denote the point where GPU memory is exhausted. Measurements were obtained by oversaturating a single-GPU system and issuing backbone-only requests to isolate the memory overhead of adapter weights. Minor variations in ITL across adapter sizes are nevertheless observed, likely due to additional operations triggered in vLLM when adapters are activated, even if unused.

Each iteration of the implemented loop is modeled as a state transition function, where the system state, comprising pending and running requests, loaded adapters, and available GPU memory, is programmatically coded and updated. These transitions are implemented by three components (scheduler, adapter cache, and model), depicted in Fig. 3, which replicate the core logic of the real system. These components integrate lightweight predictive performance models (detailed in Section 5.2) that provide latency estimates for the time-intensive tasks of each step (i.e., model forward pass). This design enables the DT to approximate the temporal evolution of the system without executing the underlying computationally intensive tasks, thereby achieving significantly faster execution than a real system and eliminating the need for GPU resources.

As depicted in Fig. 3, each simulation step begins by injecting new request arrivals according to the simulated time and the workload arrival distribution. Requests are then passed to the scheduler, which updates the active batch by removing completed requests and admitting new ones. Following the design of vLLM, a greedy KV-cache allocation strategy is applied. The updated batch is subsequently processed by the adapter cache and model components, which emulate adapter loading/unloading as well as the model forward pass, respectively.

As a standard Digital Twin, the DT requires the same inputs as the real system to perform the simulation. These include the execution duration and detailed workload characteristics, namely the arrival time of each request, the target adapter, adapter size, request input length, and the configured GPU A_{max} value. Unlike the real system, however, the DT does not internally derive the amount of output tokens per request. Instead, the expected output length must also be provided as an input parameter. Nevertheless, our evaluation shows that using the average output length across requests yields sufficiently accurate performance estimates while remaining practical to approximate in production environments. By reproducing all major execution stages of a real serving system, the DT can generate a wide range of performance metrics, including throughput, ITL, and TTFT.

5.1. Performance analysis

We describe the four main overheads that we encountered when working with adapters. This exploration represents the foundation for the design and implementation of the DT and its predictive performance models. Experiments setup is described in Section 8.1.

5.1.1. Increased memory usage

As introduced, storing adapter weights in GPU memory reduces the capacity available for the requests KV-cache, which limits the batch size and thus the throughput. Fig. 4 shows this decrease in both batch size and throughput, which appears earlier and more sharply for larger models and adapters due to their higher memory footprint. Unexpectedly, throughput decays exponentially rather than linearly as batch size. This discrepancy arises from a broader trait of LLM serving unrelated to adapter processing. As reported in several works, increasing the batch size beyond a certain point leads to diminishing returns in throughput—a phenomenon known as the throughput plateau [27, 28, 2]. This implies that, for example, with Llama-2-7B although the first 100 loaded adapters reduce the batch size, their impact on throughput is negligible, a characteristic that, to the best of our knowledge, we are the first to report. Lastly, the rightmost plot of Fig. 4 presents the relationship between ITL and batch size, which, consistent with prior reports for generic AI models [41, 29], exhibits a linear scaling trend.

Insight. Each loaded adapter affects throughput, depending on model and adapter size, but this impact may fade due to the throughput plateau.

5.1.2. Increased computational workload

Fig. 5 shows the throughput slowdown and ITL overhead caused by the additional computation and GPU internal transfers required to process adapters instead of only the backbone LLM. Both metrics increase approximately linearly with the number of adapters, except for the sharp drop from zero to one adapter. This drop corresponds to the shift from a backbone-only execution to

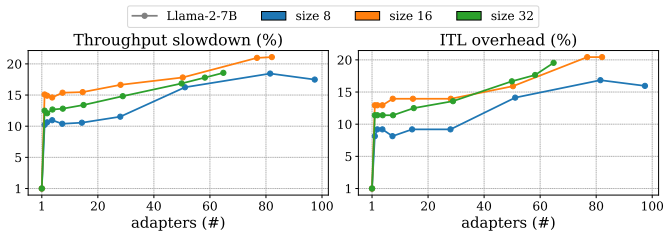


Figure 5: Throughput slowdown and ITL overhead for increasing adapters across three adapter sizes. Results are shown for Llama-2-7B and relative to backbone-only execution. To avoid the impact of adapter weights, we fix the batch size and number of loaded adapters within each line. Lines are shorter for larger adapter sizes due to their reduced achievable batch size, which limits the maximum number of runnable adapters.

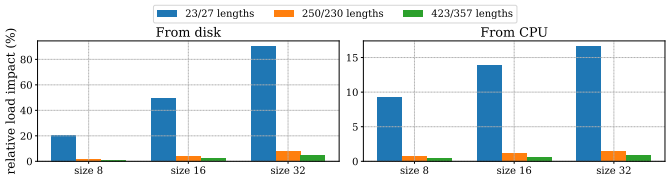


Figure 6: Loading times for varying adapter sizes, shown relative to request latency across three input/output lengths for Llama-2-7B and storage type. Request latency is computed as $TPOT * (output\_tokens - 1)$, where TPOT is the time per output token.

one that must compute both backbone and adapter activations, including the additional GPU data transfers involved. Beyond this point, computation and transfer overheads from multiple adapters is partially parallelized. Adapter size has a limited impact, with size 32 occasionally yielding lower overhead than size 16.

Insight. Serving adapters introduces computational overhead in both throughput and ITL compared to backbone-only execution, increasing linearly with the number of adapters.

5.1.3. Loading time

Fig. 6 presents the loading time to GPU memory relative to request latency, distinguishing between loading from disk and from CPU memory. Larger adapters introduce greater overhead due to their increased size, and disk loading is on average 70% slower than CPU loading. Request length also strongly affects the relative impact: for small requests, CPU loading adds 7–16% latency depending on adapter size, whereas for longer requests the overhead falls below 2%. This reduction occurs because the fixed cost of loading becomes negligible compared to the computation time of longer requests.

Insight. Loading overhead is significant only for short requests and can be largely mitigated by preloading adapters into CPU memory.

5.1.4. Scheduler

Fig. 7 shows the relative impact of the scheduler component across varying numbers of adapters and configured

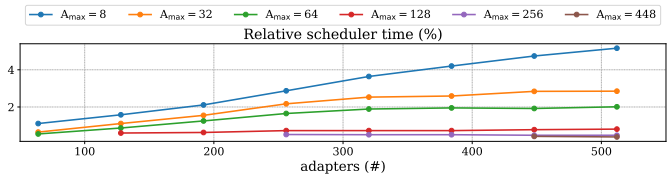


Figure 7: Scheduler time relative to the average per-step execution time, as a function of the number of adapters and configured A_{max}.

A_{max}. When the number of adapters is high but A_{max} is small, scheduling accounts for nearly 6% of execution time; in all other cases, its impact is negligible. This behavior is specific to the vLLM implementation rather than to LLM adapter serving in general. It occurs because vLLM iterates over all pending requests to select those admissible to the active batch; with small A_{max}, the scheduler must scan a larger fraction of requests to find those corresponding to loaded adapters, as requests from unloaded adapters cannot be processed because A_{max} has already been reached.

Insight. The scheduler component of vLLM may introduce an overhead when A_{max} is small relative to the number of adapters.

5.2. Predictive performance models

As illustrated on the right side of Fig. 3, the Digital Twin relies on four predictive performance models, summarized in Equation (1), which are invoked at each step of the reproduced continuous batching loop,. Three of these models estimate task-specific latency—Lat_{sched}, Lat_{load} and Lat_{model}—while the fourth, Mem_{max}, serves as an auxiliary estimator used by the scheduler to determine the maximum number of requests that can be batched together under the memory pressure introduced by adapter weights (see Section 5.1.1). Specifically, Mem_{max} takes A_{max} and S_{max} as inputs and outputs the maximum number of request tokens that can fit within GPU memory (T_{max}). This estimator is derived directly from profiled data, analogous to the leftmost plot in Fig. 4, but estimates the achievable token count rather than batch size. Although an analytical formulation could be derived from backbone and adapter sizes, using empirical profiling results proved more straightforward and equally accurate.

Lat_{sched} is used by the scheduler to estimate the time spent in the original vLLM scheduling component. It takes as input the batch size (B), the number of pending requests (R_P), the number of unique adapters currently in the batch (A_B), and the total number of adapters being served (A). Its formulation follows a analytical expression parameterized using profiling data collected from the real system scheduler under diverse workload conditions. The first two terms represent the baseline scheduling behavior: an iteration over the active batch to detect completed requests and manage KV allocation, followed by a partial iteration over pending requests to assess their eligibility for inclusion. The final term models the overhead described in

Section 5.1.4, which accounts for the additional execution time caused by scheduler-specific inefficiencies.

Lat_{load} estimates the time required to load adapters during swapping (L) as a function of their size S (see Section 5.1.3), assuming unloading is negligible. Similar to Mem_{max}, this model is derived directly from profiled data, specifically the measurements used to generate Fig. 6. Loading is modeled from CPU memory, since disk loading is treated as a one-time initialization cost.

Finally, Lat_{model} estimates the latency arising from GPU transfers and model forward pass. It is decomposed into two components: the backbone LLM latency ($Lat_{backbone}$) and the additional computational overhead from serving adapters ($Overhead_A$). The backbone latency is modeled as a linear function of batch size, consistent with prior work [41, 29] and supported by the profiling results in the rightmost plot of Fig. 4. The adapter-related overhead is modeled as a linear function of the number of adapters, following the findings in Fig. 5, which differs from previous modeling [18]. The constants for both components are parametrized from the profiling data shown in the referenced figures.

$$\begin{aligned} Mem_{max}(A_{max}, S_{max}) &= T_{max} \\ Lat_{sched}(B, R_P, A_B, A) &= K_1 B + K_2 R_P + K_3 R_P \frac{A_B}{A} \\ Lat_{load}(S_A) &= L_A \\ Lat_{model}(B, A) &= Lat_{backbone} * Overhead_A \\ &= (K_4 B + K_5) * (K_6 A + K_7) \end{aligned} \tag{1}$$

where all constants K_x are parametrized with profiled data on the chosen backbone LLM, adapters, and hardware, via non-linear least squares fitting [34].

6. ML modeling

The ML phase produces two predictive models that estimate, for a single-GPU setting, (i) achievable throughput and (ii) starvation risk for a given workload, adapter placement and A_{max} configuration. These estimators are repeatedly invoked by the greedy caching algorithm to guide distributed placement decisions. We adopt classical machine learning techniques, including Random Forests and Support Vector Machines, which provide strong predictive accuracy with moderate computational overhead. An optional refinement phase further reduces inference latency and improves interpretability, both critical for scalable and accountable production deployment.

The models are trained on a performance dataset generated by executing the DT across a wide range of workload and device configurations. Each simulated scenario contributes one sample consisting of: (i) a feature vector encoding workload and device characteristics, (ii) the DT-estimated throughput, and (iii) a binary starvation indicator. Starvation is defined as the condition where DT-estimated throughput falls below 90% of the total incoming token rate. Although the DT supports arbitrary workload distributions, training data is restricted to long-term workload patterns that can be anticipated in advance. In practice, real production traces are approximated as per-adapter Poisson processes, with requests characterized by global average input and output token lengths, as these quantities are easier to estimate in production settings and yield sufficiently accurate results.

The feature vector characterizes both the expected workload and the GPU configuration, comprising: the number of served adapters A; the sum and standard deviation of adapter Poisson arrival rates; the maximum, mean, and standard deviation of adapter sizes; and the configured A_{max} value. Two independent models are trained: a regression model for throughput prediction and a binary classifier for starvation detection.

6.1. Refinement phase

Among all evaluated model families, tree–based models achieve the best trade-off between predictive accuracy and interpretability. We therefore further refine these models along two complementary dimensions: inference efficiency and interpretability. Starting from the best-performing trained model, we progressively reduce model complexity until obtaining a single shallow decision tree. To this end, the hyperparameter optimization process is modified to strongly penalize complex structures. Model complexity is quantified as the number of logical decision rules of the form "$\text{condition}_1 \wedge \text{condition}_2 \wedge \cdots \rightarrow$ output" into which a decision tree can be decomposed. After complexity reduction, inference performance is further optimized by extracting the learned decision logic and re-implementing it in plain Python, accelerated using Numba [17]. This approach removes framework overhead and leverages code compilation to generate efficient machine code. Overall, this refinement phase substantially reduces inference latency and improves model interpretability, at the cost of a moderate degradation in predictive accuracy.

7. Caching greedy algorithm

The greedy caching algorithm constitutes the final stage of the proposed pipeline. Its objective is to solve the adapter caching problem by minimizing the number of GPUs required to serve a given workload through appropriate adapter placement and A_{max} configuration, while avoiding starvation and memory errors. The algorithm leverages the ML models to estimate the expected throughput and starvation risk associated with candidate GPU allocations and device configurations. This problem can be viewed as a sophisticated variant of the bin packing problem [8], and is therefore NP-hard. To address it efficiently, we adopt a tailored variation of the First-Fit Decreasing (FFD) algorithm [15], adapted to the specific constraints and objectives of the adapter caching problem.

The pseudocode is shown in Algorithm 1. The algorithm takes as input the expected future workload, defined by the set of adapters to serve (A), and for each adapter $a \in A$, its expected size $s[a]$ and Poisson arrival rate $\lambda[a]$. It also receives the total number of GPUs in the system (G). The outputs are the GPU assignment of each adapter ($g[a]$) and the configuration of A_{max} per GPU ($A_{max}[g]$). If no starvation-free allocation is feasible, the algorithm raises an exception to enforce the non-starvation constraint.

Algorithm 1 Caching greedy algorithm

Input: GPUs $G = \{g_1, \ldots, g_M\}$; adapters $A = \{a_1, \ldots, a_N\}$; sizes $s[a]$; rates $\lambda[a]$
Output: Assignment $g[a]$; configuration $A_{max}[g]$

1: $g[a] \leftarrow \emptyset$ **for all** $a \in A$; $A_{max}[g] \leftarrow 0$ **for all** $g \in G$
2: $A \leftarrow$ PRIORITYSORTING$(A, \text{key} = (\lambda[a], s[a]))$
3: $A_q \leftarrow$ QUEUE(A); $G_q \leftarrow$ QUEUE(G)
4: **while** $A_q \neq \emptyset$ **do**
5: $a \leftarrow$ POPLEFT(A_q)
6: **if** $G_q = \emptyset$ **then** ERROR STARVATION
7: $g \leftarrow$ POPLEFT(G_q)
8: PROVISIONALINCLUDE$(g, a, s[a], \lambda[a])$
9: **if** REACHTESTINGPOINT(g) **then**
10: $(ok,\ alloc\_set,\ p_{\text{new}}) \leftarrow$ TESTALLOCATION(g)
11: **if** ok **then**
12: COMMITALLOCATION(g)
13: **for all** $a' \in alloc\_set$ **do** $g[a'] \leftarrow g$
14: $A_{max}[g] \leftarrow p_{\text{new}}$
15: PUSHLEFT(G_q, g)
16: **else**
17: $un\_alloc\_set \leftarrow$ ROLLBACKALLOCATION(g)
18: $A_q \leftarrow$ MERGE$(A_q,\ un_alloc_set)$
19: **end if**
20: **else**
21: PUSHLEFT(G_q, g)
22: **end if**
23: **end while**
24: **for all** $g \in G$ **with** non-tested allocation **do**
25: $(ok,\ alloc\_set,\ p_{\text{new}}) \leftarrow$ TESTALLOCATION(g)
26: **if** *not ok* **then** ERROR STARVATION
27: repeat lines 12-14
28: **end for**
29: **return** (Assignment $g[a]$, configuration $A_{max}[g]$)

The algorithm follows a greedy strategy, allocating adapters sequentially to maximize GPU packing up to the optimal point Max_{pack}. Allocation order is determined by the PRIORITYSORTING step: adapters are first sorted by size (largest first) and then by arrival rate in a zigzag order (alternating between high and low) while preserving size-based ordering. Sorting by size groups adapters with similar sizes and prioritizes larger ones first, thereby preventing newly allocated adapters from increasing S_{max} device configuration. The zigzag ordering was selected empirically, as it consistently improved throughput in our experiments. Exhaustive per-adapter testing is infeasible, so the algorithm performs provisional allocations via PROVISIONALINCLUDE until a predefined testing point is reached (REACHTESTINGPOINT). At that point, feasibility is evaluated: successful allocations are committed (COMMITALLOCATION), while failed ones are rolled back (ROLLBACKALLOCATION), merged back into the pending queue (MERGE), and retried on another GPU. In this work, testing points correspond to cumulative adapter counts [8, 16, 32, 64, 96, 128, 160, 192, 256, 320, 384]. After the main loop terminates, any remaining provisional allocations are validated and committed.

Feasibility is evaluated by the TESTALLOCATION method, which also determines the optimal A_{max} configuration. Its pseudocode is shown in Algorithm 2. The method receives as input the GPU under evaluation and its internal state, including the number of firmly allocated adapters, the set of provisionally included adapters, and their respective sizes and arrival rates. For brevity, internal state management is omitted from the pseudocodes. The method first queries the ML model for throughput predictions using MLPREDICTTHROUGHPUT, evaluating the GPU state under two configurations: the current A_{max} and the next candidate. As with adapter allocation, exhaustive exploration of all configurations is intractable. Instead, candidate A_{max} values are restricted to a predefined set returned by NEXTGPUCONFIG, which reuses the same array of adapter counts defined earlier. The configuration yielding the highest predicted throughput is selected. Subsequently, the method invokes MLPREDICTSTARVATION to check whether the selected configuration, given the current GPU internal state, would lead to starvation. If starvation is predicted, the allocation is rejected as infeasible; otherwise, the method returns the set of new allocatable adapters along with the chosen A_{max} configuration.

Algorithm 2 TESTALLOCATION(g)

Input: GPU g; internal state $= (\mathcal{A}_{\text{alloc}}, \mathcal{A}_{\text{prov}}, s[\cdot], \lambda[\cdot], p)$
Output: $(ok,\ alloc\_set,\ p_{\text{new}})$

1: $p_{\text{next}} \leftarrow$ NEXTGPUCONFIG(g)
2: $\mathcal{A}_{\text{all}} \leftarrow \mathcal{A}_{\text{alloc}} \cup \mathcal{A}_{\text{prov}}$
3: $T \leftarrow$ MLPREDICTTHROUGHPUT$(\mathcal{A}_{\text{all}}, s[\cdot], \lambda[\cdot], p)$
4: $T_{\text{next}} \leftarrow$ MLPREDICTTHROUGHPUT$(\mathcal{A}_{\text{all}}, s[\cdot], \lambda[\cdot], p_{\text{next}})$
5: **if** $T > T_{\text{next}}$ **then** $p_{\text{best}} \leftarrow p$ **else** $p_{\text{best}} \leftarrow p_{\text{next}}$
6: $starve \leftarrow$ MLPREDICTSTARVATION$(\mathcal{A}_{\text{all}}, s[\cdot], \lambda[\cdot], p_{\text{best}})$
7: **if** $starve$ **then return** (FALSE, $\emptyset, \emptyset$)
8: **return** (TRUE, $\mathcal{A}_{\text{prov}}$, p_{best})

8. Evaluation

This section evaluates the Digital Twin and ML learning phase in approximating real LLM-adapter serving behavior, and the effectiveness of the caching greedy algorithm in addressing the adapter caching problem.

8.1. Setup

Data. We sample requests from a cleaned version of the ShareGPT dataset [3], preserving their original heterogeneous input and output lengths. Nevertheless, for the data used to parametrize the DT performance models, we generate synthetic requests composed of random words to avoid bias in request content.

Metrics. In all textual references, figures, and tables, *throughput/through.* denotes the total processing rate, computed as the sum of input throughput (tokens processed as input) and output throughput (tokens generated as output).

LLM models. We use Llama-3.1-8B-Instruct [9] and Qwen2.5-7B-Instruct [38] with LoRA adapters of varying sizes derived from two HuggingFace adapters [36, 42]. As of Section 5.1, developed early in this work, we test Llama-2-7B and Llama-2-13B models [33] with LoRA adapters also based on a HuggingFace adapter [39]. In all experiments, we set S_{max} as the maximum adapter size across each scenario, following the default behavior in vLLM.

ML learning phase. We evaluate three model types: K-Nearest Neighbors (KNN), Random Forest (RF), and Support Vector Machine (SVM), all coming from the Python scikit-learn library [24]. Training is performed with 5-fold cross-validation, and hyperparameter tuning is conducted using the HalvingGridSearchCV method from scikit-learn. The details of the hyperparameter search space are reported in Appendix B.

Framework. Experiments use vLLM version *v0.8.5*, except for the analyses in Section 5.1, which was developed earlier using version *v0.5.0.post1*. In terms of implementation, we introduce minor modifications to the server components to collect auxiliary metrics and support additional arguments, and substantially update the benchmarking script to execute realistic online environments with adapters. In distributed scenarios, a different vLLM instance is deployed in each GPU with the adapter allocation provided by the placement algorithms, and requests routed according to their assigned adapter.

Hardware. Experiments are conducted on a node equipped with four NVIDIA Hopper H100 (64GB HBM2), 512GB RAM memory, and 80 CPU cores.

8.2. Digital Twin

We evaluate the accuracy of the proposed DT by comparing its predicted performance metrics against those of a real LLM-adapter serving system, assessing its ability to reproduce system behavior across a wide range of different workloads. These are generated through a Cartesian combination of two adapter-size sets—high-to-low $\{8, 16, 32\}$ and medium-to-low $\{8, 16\}$—and two Poisson arrival-rate regimes—high $\{1.6, 0.8, 0.4\}$ and low $\{0.1, 0.05, 0.025\}$ requests per second. For each workload, adapters randomly select a size and a arrival rate from the chosen sets, producing heterogeneous configurations. The number of served adapters is varied from 8 to 384, and the corresponding values of A_{max} are explored within the same range. Real-system experiments are executed for one hour per configuration, and accuracy is quantified using the Symmetric Mean Absolute Percentage Error (SMAPE) across all scenarios. Results are summarized in Table 1 under the *Predictable arrivals* setting. Two variants of DT input information regarding request lengths are evaluated. In the *Original* variant, the DT uses exact input and output token lengths from the real system. In the *Mean* variant, all requests use identical token lengths equal to the workload average. This latter configuration reflects the information typically available in practice and corresponds to the setup used during the ML learning phase.

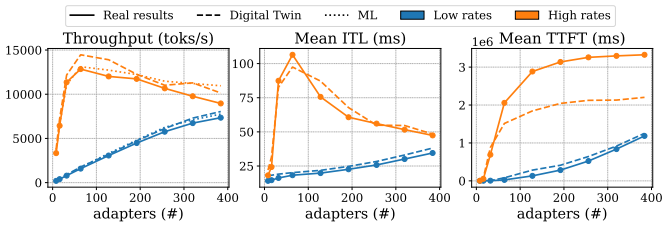


Figure 8: Comparison between DT predictions and real-system measurements for throughput, ITL, and TTFT under varying numbers of adapters and arrival rates. Experiments are conducted using adapter sizes 8 and 16 on the Qwen-2.5-7B model with $A_{max} = 8$, in the *Original* and *Mean* variants. For throughput, predictions produced by the ML model are also reported.

Overall, the DT accurately reproduces real-system behavior under predictable workloads for both models. Throughput and ITL achieve maximum SMAPE values of 5.08% and 9.63%, respectively, while TTFT exhibits higher deviation, with a maximum SMAPE of 18.95%. Although the *Original* variant consistently yields lower error, the *Mean* configuration remains comparably accurate, supporting its use in practical scenarios.

Table 2 shows DT time and resource usage. It achieves up to a 90× speedup compared to one-hour real-system executions, using only a single CPU core, no GPU, and approximately 200 MB of memory. Fig. 8 illustrates a comparison between DT predictions and real-system measurements for a subset of the tested scenarios. Consistent with the table results, throughput and ITL are closely matched, whereas TTFT shows larger deviations, particularly under higher arrival rates.

Unpredictable arrivals: We further evaluate the robustness of the DT under unpredictable arrivals. To that end, each adapter independently updates its arrival process every five minutes. At each update, the arrival distribution is randomly selected between Poisson and log-normal, and the corresponding arrival rate is randomly multiplied or divided by a factor of two. To prevent unrealistic behaviors, arrival rates are clipped within predefined bounds. This configuration generates strongly non-stationary traffic patterns, providing a challenging setting for assessing the DT's ability to reproduce system behavior under rapidly changing workloads. An example of the

Model	Req. lengths	Predictable arrivals SMAPE comparison (%)			Unpredictable arrivals SMAPE comparison (%)		
		Through.	ITL	TTFT	Through.	ITL	TTFT
Llama	Original	**2.25**	**7.09**	18.83	**2.47**	**5.04**	19.95
	Mean	4.18	9.63	**17.44**	3.66	9.92	**19.91**
Qwen	Original	**2.59**	**7.18**	18.95	5.16	9.87	20.74
	Mean	5.08	9.07	**17.92**	**5.12**	**9.22**	**20.72**

Table 1: Final evaluation of the proposed DT across all predictable (left) and unpredictable (rigth) test scenarios. Reported values correspond to SMAPE between DT predictions and real-system measurements, where lower values denote higher fidelity.

	Resource consumption		
Model	Time (s)	CPU (%)	Mem (MB)
Llama	38.94 ± 5.58	89.51 ± 1.69	202.58 ± 10.85
Qwen	39.61 ± 5.55	90.33 ± 1.30	204.32 ± 9.29

Table 2: Execution time and resource consumption results of the DT across all tested scenarios.

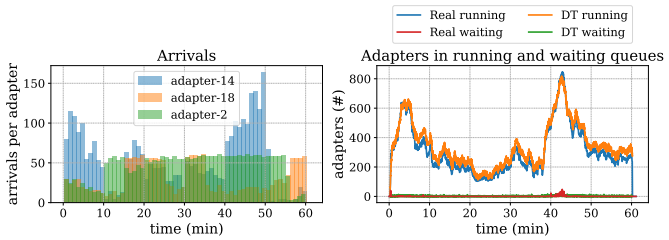


Figure 9: Execution with Llama-3.1-8B under initial high arrival rates (1.6, 0.8, 0.4) for 32 adapters in the unpredictable regime. (Left) Non-stationary arrival traces for randomly sampled adapters. (Right) Comparison of running and waiting requests over time between the DT and the real system.

resulting arrival traces is shown in the left plot of Fig. 9.

We follow the same evaluation methodology as in the predictable-arrival experiments, where the sampled arrival rates only determine the initial values before applying the described transformations. The resulting SMAPE values are shown in the right part of Table 1. Overall, the DT accurately reproduces system behavior under unpredictable arrival patterns, achieving error levels comparable to those observed for predictable workloads, albeit with a slightly higher average error, which is expected given the increased complexity of the scenario. The right plot of Fig. 9 illustrates the evolution of running and waiting requests over time for both the DT and the real system in a specific scenario. The close alignment between the two traces highlights the DT's ability to replicate scheduler dynamics, KV-cache allocation, adapter loading, and computation latency under highly dynamic traffic conditions.

8.3. ML modeling

Leveraging the DT for rapid data generation, we train the proposed ML models over an extended and diverse set of workload scenarios. Workloads are generated through a Cartesian product of adapter sizes and arrival rates. Instead of evaluating only a small number of predefined sets, we construct all combinations of three values drawn from the adapter-size set $\{8, 16, 32\}$ and the arrival-rate set $\{3.2, 1.6, 0.8, 0.4, 0.1, 0.05, 0.025, 0.0125, 0.00625, 0.003125\}$. For each workload configuration, we further vary both the number of served adapters and the device configuration parameter A_{max} within the range 8 to 384. This process results in a large and heterogeneous training dataset that captures a wide range of workload conditions. The time required to generate the synthetic dataset and train the ML is reported in Table 3. Both processes exhibit execution times suitable for offline execution in production environments. Notably, the execution times for Qwen are higher, as the employed model variant is smaller than the Llama counterpart, allowing a greater number of adapters to be packed without incurring memory errors, resulting in a larger training dataset and increased processing time.

Table 4 summarizes the prediction accuracy of the three evaluated ML model types (KNN, RF, and SVM), evaluated against the same real-system executions employed for the validation of the DT. Throughput prediction accuracy is quantified using SMAPE, while starvation detection performance is evaluated using macro-averaged F1 score across all scenarios.

Overall, the ML models achieve high predictive fidelity, particularly for starvation detection. Throughput prediction errors remain below 8%, slightly higher than those obtained using the DT, as expected given the additional abstraction introduced by the learning process. Fig. 8 additionally reports throughput predictions produced by the best-performing RF model, showing close agreement with real-system measurements. Prediction latency, also reported in Table 4, remains below 0.3 ms for all models except SVM, representing a substantial improvement over DT execution time.

Refinement phase: We further extend the ML learning phase with the refinement procedure described in Sec-

Model	DT dataset generation	ML training time	
		Through.	Starvation
Llama	6 hours	270 seconds	197 seconds
Qwen	9 hours	300 seconds	293 seconds

Table 3: Time required to generate the dataset used for training the ML models with the DT when parallelized across 80 CPUs on a single node, and time required to train the RF models for throughput and starvation prediction (including hyperparameter search), which are the ones mostly selected for online placement decisions.

Model	Estimator	Throughput		Starvation	
		SMAPE (%)	Time (ms)	F1 (macro)	Time (ms)
Llama	KNN	4.52	**0.21**	0.95	0.19
	RF	**4.39**	0.25	0.95	0.16
	SVM	6.84	1.70	**0.98**	**0.02**
Qwen	KNN	**5.28**	**0.15**	**0.99**	0.19
	RF	**5.28**	0.21	**0.99**	0.22
	SVM	7.46	2.24	0.93	**0.05**

Table 4: Final evaluation results of the proposed ML model across the three model types and both LLMs. It evaluates throughput estimation (with SMAPE) and starvation detection (with macro-F1), and reports the average prediction time in milliseconds.

tion 6. Table 5 reports results for the original RF model, the simplified shallow decision tree (*Small Tree*), and its Numba-optimized implementation (*Small Tree***).

The refinement process yields substantial improvements in both inference efficiency and interpretability. The simplified models use at most 32 decision rules, compared to at least 3.79e3 in the baseline RF. Inference latency is reduced by over 69× for the shallow tree and up to 2120× for the Numba-optimized implementation, achieving inference times below 100 ns per prediction in some occasions. These gains are obtained at the cost of reduced predictive accuracy. On average, throughput estimation exhibits a 6.74% increase in SMAPE, while starvation detection experiences a decrease of 0.025 in macro-F1 score. This trade-off remains acceptable in scenarios where inference speed and model interpretability are primary requirements.

Beyond performance improvements, the simplified models enable direct interpretability and extraction of actionable insights. Appendix C presents the learned decision trees for both starvation and throughput prediction. For instance, we can extract that starvation is unlikely when the aggregate incoming rate remains below 23.52 tokens/s, and that excessively large A_{max} values, above 144, can negatively impact overall system throughput.

8.4. Caching decisions

We evaluate the proposed pipeline as a solution to the adapter caching problem, with the greedy algorithm acting as the final decision stage. We first analyze single-GPU scenarios to demonstrate throughput maximization and identification of the per-GPU optimal packing point (Max_{pack}). We then extend the analysis to distributed settings with four GPUs, showing how consistently reaching Max_{pack} improves overall GPU efficiency. Finally, we evaluate an alternative configuration of the pipeline in which the optimization objective is shifted from resource efficiency to latency minimization.

We derive the request arrival pattern from the 2025 Azure multimodal model inference trace [26], which contains data collected between October 15th and 22nd, 2024. For each subplot presented in this section, we randomly sample a single day and a pair of consecutive hours from the trace, and uniformly distribute requests across 1280 adapters. The placement algorithms are provided with the average per-adapter arrival rate observed during the first hour (i.e., modeled as Poisson process), and are tasked with determining the optimal placement for the subsequent hour. Evaluation then replays the actual arrivals from the second hour on a real LLM-adapter serving system using the provided placements. We consider two settings for adapter sizes: (i) sizes randomly drawn from {8, 16}, and (ii) sizes randomly drawn from {8, 16, 32}. This distinction is important, as in the latter case S_{max} must be set higher, significantly reducing the available memory for serving incoming requests.

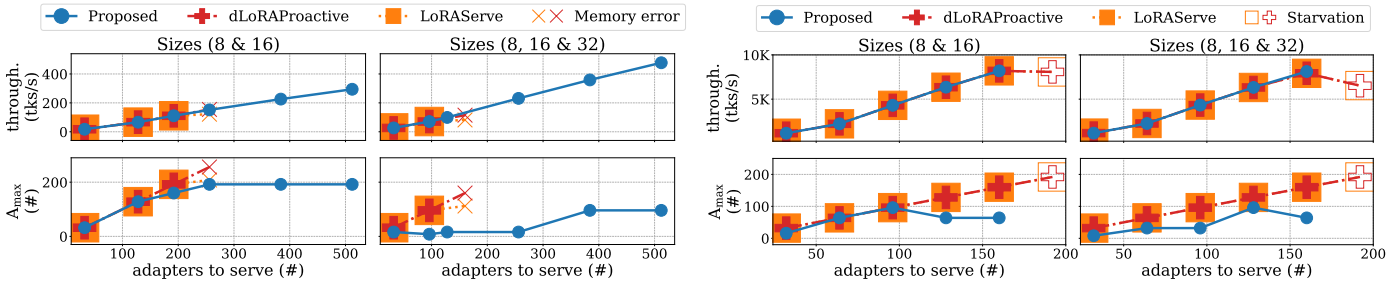
Baselines. We compare our approach against the two most closely related methods:

- **dLoRAProactive** [37]: We use the original dLoRA codebase to replicate its proactive adapter placement strategy. Other components of dLoRA are excluded, as they are orthogonal to the adapter placement problem considered in this work. When an adapter is replicated across multiple GPUs, incoming requests are randomly distributed among them. As in the original implementation, A_{max} is set to the number of adapters served per GPU, and no explicit constraint is imposed on the maximum workload that can be assigned to each GPU.

- **LoRAServe** [14]: As the official implementation is not publicly available, we re-implement the method based on the algorithmic description provided in the paper. When adapters are replicated across multiple GPUs, we use the output probabilities of their algorithm to distribute incoming requests accordingly. Since A_{max} is not specified, we set it to the number of adapters served per GPU, which is the most straightforward choice. Consistent with their description, we use the maximum profiled throughput for a single adapter as the upper bound on the workload assignable to each GPU.

Model		No. rules	Throughput		No. rules	Starvation	
			SMAPE (%)	Time (ms)		F1 (macro)	Time (ms)
Llama	RF	7.13e6	**4.39**	0.25	4.44e4	**0.95**	0.16
	Small Tree	**32**	10.25	3.24e-3	**16**	0.92	3.25e-3
	Small Tree**	**32**	10.25	**1.13e-4**	**16**	0.92	**8.30e-5**
Qwen	RF	6.33e6	**5.28**	0.21	3.79e3	**0.99**	0.22
	Small Tree	**16**	12.89	2.52e-3	**15**	0.97	3.21e-3
	Small Tree**	**16**	12.89	**9.36e-5**	**15**	0.97	**10.5e-4**

Table 5: Evaluation results of the ML models obtained by progressively simplifying the best-performing RF model into a shallow, interpretable decision tree (Small Tree), and its Numba-optimized implementation (Small Tree**).



(a) Llama

(b) Qwen (arrivals ×100)

Figure 10: (Top) Achieved throughput and (bottom) configured A_{max} on a single-GPU system by our method and the two baselines, under both adapter size settings and models. In the Qwen case, Azure arrivals are scaled by a factor of ×100 to test a higher-load scenario. Each curve is shown up to the point where the corresponding method deems the placement infeasible, or encounters starvation or memory errors.

Our approach, denoted as *Proposed*, combines the caching greedy algorithm with the best-performing ML models identified in Table 4, which are trained on DT data. Models obtained after the refinement phase are also evaluated in the distributed setting, labeled as *ProposedFast*.

8.4.1. ***Maximizing per-GPU utilization***

Figure 10 reports the throughput achieved and configured A_{max} by the proposed placement of our method and the baselines as the number of adapters to serve increases. The left subplot show Llama under the two adapter size settings, while the right subplot presents Qwen under the same setup, with Azure trace arrivals scaled by ×100 to represent a higher-load scenario.

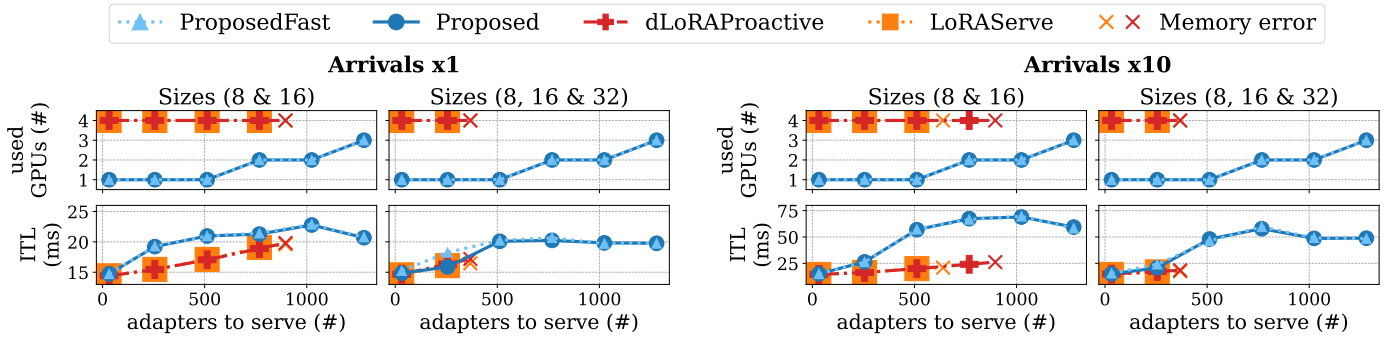
We observe two distinct regimes. First, under low arrival rates (e.g., Llama in Figure 10a), GPU memory for adapter weights becomes the primary bottleneck as the number of adapters increases. Both baselines fail to properly control A_{max}, producing placements that exceed the available GPU memory for adapter weights and result in memory errors. In contrast, our method initially allows high parallelism, with A_{max} values close to the number of served adapters, but progressively constrains it as the number of adapters grows, particularly in the setting that includes larger adapter sizes (i.e., 32). This prevents memory violations and enables serving over 500 adapters, compared to fewer than 200 for the baselines, achieving more than 2× higher throughput.

The second regime arises under high arrival rates (e.g., Qwen in Figure 10b). In this setting, the main bottleneck shifts to the memory required for request KV values, driven by the large number of concurrent requests. As the number of adapters increases, insufficient KV capacity leads to request starvation in both baselines. Our method, however, detects such infeasible configurations and stops before generating placements that would incur starvation. Instead, it limits the per-GPU workload to the maximum feasible throughput (Max_{pack}), ensuring each GPU operates within its capacity. Excess demand is deferred to other GPUs, as discussed in the next section, thereby preventing starvation while maximizing per-GPU utilization.

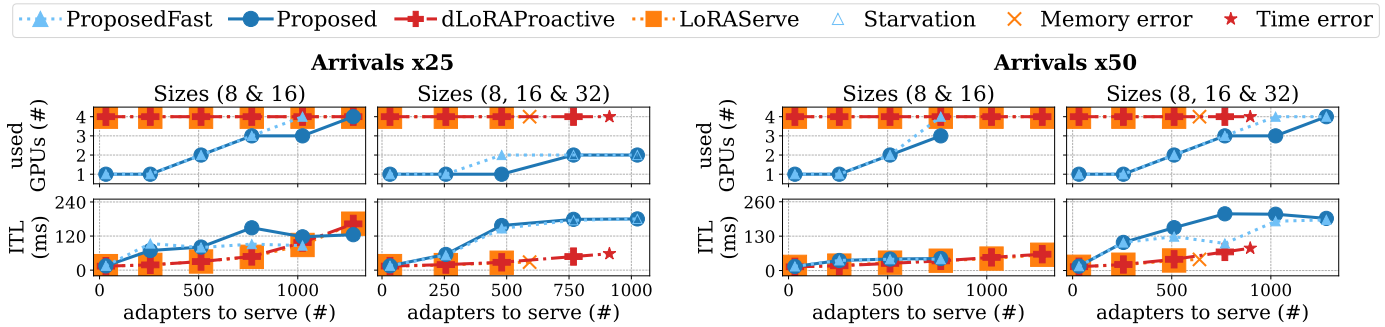
8.4.2. ***Efficient utilization of GPU resources***

Figure 11 reports the number of GPUs utilized and the achieved ITL for our approach and the two baselines in a distributed setting with four GPUs. Results are presented for both models and adapter settings under increasing arrival load conditions (from ×1 to ×50), while scaling the number of served adapters from 32 to 1280, following the initial partitioning of the Azure trace.

The baseline methods exhibit memory errors, as they do not properly adapt the A_{max} configuration to the workload characteristics, consistent with the behavior previously observed in Figure 10a. As a result, they are unable to scale



(a) Llama (arrivals ×1 and ×10)



(b) Qwen (arrivals ×25 and ×50)

Figure 11: Number of GPUs required and ITL achieved by each method on a 4-GPU system, for both models under increasing arrival load as the number of served adapters scales from 32 to 1280. Each curve is shown up to the point where the corresponding method deems the placement infeasible or encounters starvation or memory errors. Time-limit failures for dLoRA occur when its placement algorithm does not complete within the imposed timeout of two hours (while the scenario evaluation itself lasts one hour). These failures arise for large adapter counts and model sizes, which may be consistent with the original work, where not evaluation was done beyond 128 adapters.

to higher adapter counts. In contrast, our method successfully serves a substantially larger number of adapters. For example, in the leftmost Llama subplot, it supports up to 1280 adapters, whereas the baselines are limited to 768. These limitations arise earlier for Llama (8B) than for Qwen (7B) due to the larger size of the former, which results in a higher memory footprint per adapter.

The key observation is that our method leverages estimates of near-peak GPU utilization, previously seen in Figure 10b, to maximize adapter packing per GPU. This enables a significant reduction in the number of GPUs required to serve a given workload. While the baselines utilize all available GPUs, our approach reduces GPU usage by an average of 2.4 GPUs (60%) across the evaluated scenarios. For instance, in the rightmost Qwen configuration, our method serves 256 and 768 adapters using only 2 and 3 GPUs (out of 4), respectively, while still serving the workload without starvation. An exception is observed in the center-right Qwen configuration, where peak utilization is underestimated, leading to incorrectly determining that workloads beyond 768 adapters cannot be accommodated within the available resources.

These efficiency gains come at the cost of increased latency, as reflected in the ITL metric. This is expected, as higher adapter counts increase request load and batch size, thereby raising latency, consistent with Figure 4. Ultimately, this trade-off depends on the desired optimization objective, in our case, the focus is on maximizing GPU efficiency rather than minimizing latency (see the following section for a latency-oriented variant of our approach).

Finally, Table 6 reports the average execution time required to generate a placement. While the *Proposed* approach incurs higher computational overhead than the baselines, its latency remains suitable for periodic allocation updates. The *ProposedFast* variant, which incorporates a refinement phase in the underlying ML models, achieves comparable execution cost to LoRAServe and is approximately two orders of magnitude faster than *dLoRAProactive*, with an average latency of 3–4 ms per placement. Moreover, results in Figure 11 show that *ProposedFast* maintains performance close to *Proposed*, delivering a substantial improvement in GPU resource efficiency over the baselines. However, this speedup comes at the cost of less stable predictions, as illustrated by the left-center Qwen subplot, it may produce placements that incur starvation, indicating a trade-off between model performance and execution cost.

Method	time(s)	
	Llama	Qwen
Proposed	1.590	1.785
ProposedFast	0.004	0.004
LoRAServe	0.003	0.004
dLoRAProactive	0.654	1.013

Table 6: Average execution time per placement for the baselines and the two variants of our method, *Proposed* and *ProposedFast*, in the distributed scenario.

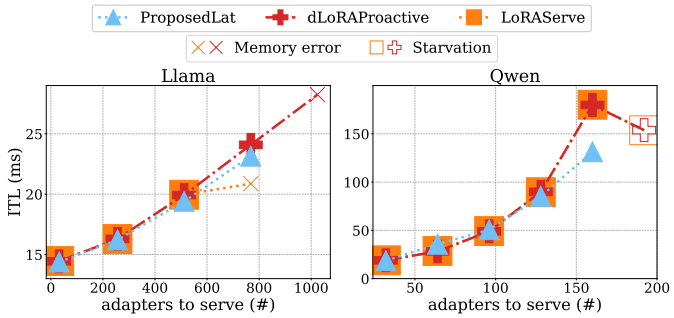


Figure 12: ITL obtained for two instances from the previous evaluations, comparing the latency-oriented variant *ProposedLat* against the two baselines. Specifically, (left) Llama with arrivals scaled by ×10 and adapter sizes 8 and 16 in the distributed scenario; and (right) Qwen with arrivals scaled by ×100 and adapter sizes 8, 16, and 32 in the single-GPU scenario.

8.4.3. ***Latency oriented***

To assess whether the proposed pipeline can be adapted to alternative objectives beyond GPU efficiency, we implement a proof-of-concept variant targeting latency minimization. This prototype, denoted *ProposedLat*, reuses the learned ML models but replaces the throughput-oriented greedy algorithm with a latency-oriented heuristic. Specifically, it assigns adapters sequentially to the GPU with the lowest aggregated arrival rate and configures A_{max} as the number of adapters served on each GPU. After all adapters are assigned, the resulting allocation is validated using the learned ML models. Allocations predicted to yield a throughput lower than the incoming token rate, or to incur memory errors, are deemed infeasible. Fig. 12 presents its results alongside the two baselines, for two instances from the preceding evaluations.

As shown, *ProposedLat* achieves latency comparable to the baselines as it also utilizes all available GPUs by design. The key distinction lies in the integration of the learned ML models, which enables to avoid infeasible allocations that lead to memory errors (left) or request starvation (right). This makes *ProposedLat* more suitable for production environments and highlights the practical value of the proposed contributions beyond maximizing GPU efficiency. Notably, *ProposedLat* employs a simple A_{max} configuration, which could be further improved as in the default *Proposed* strategy.

9. Discussion

Our evaluation demonstrates that the proposed pipeline significantly enhances GPU efficiency in distributed LLM-adapter serving systems, achieving an average reduction of 60% in the number of required GPUs compared to baselines (Fig. 11). This improvement stems from systematically identifying the Max_{pack} operating point and the corresponding A_{max} configuration that maximize per-GPU throughput while avoiding starvation and memory errors (Fig. 10). Consequently, the system can serve the target workloads using a reduced subset of devices, thereby freeing the remaining GPUs for additional tasks or potential energy savings. The online computation time of the pipeline, reported in Table 6, meets the requirements of periodic reconfiguration under predictable workload conditions. Furthermore, the optimized variant, *ProposedFast*, refines the underlying machine learning models into shallow, interpretable decision trees, reducing execution time by several orders of magnitude while enhancing transparency in the decision-making process. This efficiency gain is achieved at the cost of a slight increase in allocation instability, a trade-off that is acceptable in scenarios where rapid placement estimation is prioritized over predictive accuracy.

As anticipated, increasing per-GPU packing leads to higher latency compared to baseline approaches, primarily due to larger effective batch sizes. Nevertheless, we further evaluate a latency-oriented variant of the proposed pipeline, *ProposedLat* that achieves latency comparable to baseline methods while maintaining robustness against starvation and memory errors. These results indicate that the proposed pipeline can also be beneficial for alternative optimization objectives beyond resource efficiency.

Aside its role in the proposed pipeline, the DT constitutes an additional contribution. To the best of our knowledge, it is the first digital twin specifically tailored to LLM-adapter serving. As shown in Table 1 and Figs. 8 and 9, it closely reproduces real-system performance, particularly throughput, across both predictable and unpredictable workloads, while operating significantly faster and at substantially lower computational cost (Table 2). This enables not only rapid synthetic dataset generation, but also broader applications such as scheduling optimization and server configuration exploration.

Finally, Section 5.1 provided an in-depth characterization of LLM-adapter serving behavior, detailing the impact of the four principal overheads introduced by adapters. Beyond being the basis for the design and implementation of the proposed DT, this analysis offers independent value by exposing key system-level relationships, such as the connection between the adapter memory footprint overhead and the throughput plateau.

9.1. Limitations and future work

Even though the evaluation is conducted using the ShareGPT dataset, which exhibits a heterogeneous distribution of input and output sequence lengths, reliance on a single dataset limits the generalization of the results to workloads with substantially different length characteristics. Future work will extend dataset generation to encompass a broader range of sequence-length distributions, enabling the placement strategy to better generalize to more diverse and heterogeneous serving conditions.

Additionally, an important direction for future research is the study of online retraining mechanisms for both the DT and the associated ML models, allowing the system to dynamically adapt to evolving workload patterns observed in production traces. Finally, we will investigate the underlying causes of the sole exception in which the *Proposed* method exhibits overly conservative behavior (right-center subplot of Qwen in Fig. 11), classifying certain workloads as infeasible despite being successfully served by the baselines. This limitation is likely attributable to gaps in the training data generated by the DT, which should be expanded. In particular, the initial evaluation of this work tested exclusively on Poisson arrival processes and did not account for the Azure trace characteristics considered in the current evaluation.

10. Conclusions

We presented a data-driven pipeline to address the adapter caching problem, improving GPU efficiency in distributed LLM-adapter serving through workload-aware adapter placement. We evaluated the approach under heterogeneous and diverse workloads against state-of-the-art baselines. The results demonstrate that the pipeline effectively maximizes per-GPU throughput while minimizing the number of GPUs required to serve the target workloads, without incurring starvation or memory errors. The pipeline can be integrated into production systems to periodically update placements for predictable workload patterns, reducing the amount of needed hardware. The freed GPUs can be reassigned to other workloads to improve overall system efficiency or powered down to reduce energy consumption. Central to this approach is a Digital Twin for LLM-adapter serving, which closely reproduces real-system performance at low cost, enabling efficient ML training with performance data and supporting additional optimization tasks beyond this work.

Acknowledgment

This work has been partially financed by the EU-HORIZON MSCA programme under grant agreement EU-HORIZON MSCA GA.101086248. Also, it has been partially financed by Generalitat de Catalunya (AGAUR) under grant agreement 2021-SGR-00478, by Severo Ochoa Center of Excellence CEX-2021-001148-S-20-3, and by the Spanish Ministry of Science (MICINN), the Research State Agency (AEI) and European Regional Development Funds (ERDF/FEDER) under grant agreement PID2024-160996OB-I00, MICIU/AEI/10.13039/501100011033/FEDER, UE.

The authors used OpenAI's GPT model to assist with text rephrasing and stylistic refinement. All content was originally written and subsequently reviewed and validated by the authors.

Appendix A. S-LoRA

We include a brief analysis using the S-LoRA framework [31] to show that the adapter caching problem is not specific to vLLM and also arises in other serving systems. Fig. A.13 reports the Max_{pack} point in S-LoRA across different arrival rates, for a fixed adapter size and request-length distribution, following the same methodology as the middle plot of Fig. 1 for vLLM. Notably, throughput degradation plateaus as the number of adapters increases, whereas vLLM exhibits a more pronounced decline, highlighting the impact of S-LoRA's design choices. Nevertheless, identifying Max_{pack} remains necessary to determine how many adapters can be allocated per GPU without triggering starvation, and the throughput level at which Max_{pack} occurs still varies across workloads, with additional variability expected when changing adapter sizes and request-length distributions.

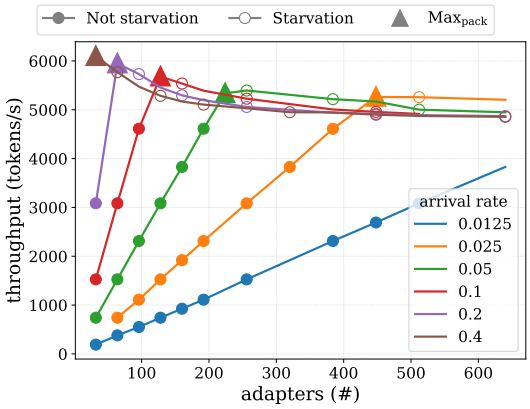


Figure A.13: Throughput of S-LoRA on Llama-2-7B under varying adapter arrival rates, using 32-sized adapters and fixed request lengths (250 input tokens, 231 output tokens).

Appendix B. ML hyperparameter search space

Hyperparameter optimization in the ML learning phase is performed using `HalvingGridSearchCV` with 5-fold cross-validation. Distinct search spaces are defined for the two tasks, throughput regression and starvation classification, using the corresponding scikit-learn model classes. The evaluated values for each search are listed below.

Throughput (Regression). a) Random Forest (`RandomForestRegressor`): `n_estimators` {32,128,256}, `max_depth` {None,5,10,20}, `min_samples_split` {2,5,10,20}, `criterion` {squared_error, absolute_error, friedman_mse, poisson}, `min_samples_leaf` {1,2,5,10,32,128}, `max_features` {auto,sqrt,log2}. b) SVM (`SVR`): `C` {0.1,1,10,100,1000,10000}, `kernel` {linear,poly,rbf,sigmoid}, `epsilon` {0.1,0.5,1,5}, `degree` {2,3,4,5}, `gamma` {scale,auto,0.01,0.1,1,10}, `coef0` {0,0.1,0.5,1}. c) KNN (`KNeighborsRegressor`): `p` {1,2}, with fixed `n_neighbors`=1, `leaf_size`=8, `weights`=uniform, `algorithm`=kd_tree.

Starvation (Classification). a) Random Forest (`RandomForestClassifier`): `n_estimators` {32,128,256}, `max_depth` {None,5,10,20}, `min_samples_split` {2,5,10,20}, `criterion` {gini,entropy,log_loss}, `min_samples_leaf` {1,2,5,10,32,128}, `max_features` {None,sqrt,log2}. b) SVM (`SVC`): `C` {0.1,1,10,100,1000,10000}, `kernel` {linear,poly,rbf,sigmoid}, `degree` {2,3,4,5}, `gamma` {scale,auto,0.01,0.1,1,10}, `coef0` {0,0.1,0.5,1}. c) KNN (`KNeighborsClassifier`): `p` {1,2}, with fixed `n_neighbors`=1, `leaf_size`=8, `weights`=uniform, `algorithm`=kd_tree.

Appendix C. Derived lightweight tree estimators

Fig. C.14 depicts two of the shallow trees resulting from the refinement phase described in Section 6.

References

[1] Agrawal, A., Kedia, N., Mohan, J., Panwar, A., Kwatra, N., Gulavani, B.S., Ramjee, R., Tumanov, A., 2024a. Vidur: A large-scale simulation framework for llm inference. Proceedings of Machine Learning and Systems 6, 351–366.

[2] Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B., Tumanov, A., Ramjee, R., 2024b. Taming throughput-latency tradeoff in llm inference with sarathi-serve, in: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pp. 117–134.

[3] anon8231489123, 2023. Clean sharegpt dataset. URL: https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

[4] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. Advances in neural information processing systems 33, 1877–1901.

[5] Brüel-Gabrielsson, R., Zhu, J., Bhardwaj, O., Choshen, L., Greenewald, K., Yurochkin, M., Solomon, J., 2024. Compress then serve: Serving thousands of lora adapters with little overhead. arXiv preprint arXiv:2407.00066 .

[6] Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., Krishnamurthy, A., 2024. Punica: Multi-tenant lora serving. Proceedings of Machine Learning and Systems 6, 1–13.

[7] Cho, J., Kim, M., Choi, H., Heo, G., Park, J., 2024. Llmservingsim: A hw/sw co-simulation infrastructure for llm inference serving at scale, in: 2024 IEEE International Symposium on Workload Characterization (IISWC), IEEE. pp. 15–29.

[8] Garey, M.R., Johnson, D.S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco.

[9] Grattafiori, A., et al., 2024. The llama 3 herd of models. URL: https://arxiv.org/abs/2407.21783, arXiv:2407.21783.

[10] Guo, D., Rush, A., Kim, Y., 2021. Parameter-efficient transfer learning with diff pruning, in: Proceedings of the 59th annual meeting of the association for computational linguistics and the 11th international joint conference on natural language processing (volume 1: Long papers), pp. 4884–4896.

[11] Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., Gelly, S., 2019. Parameter-efficient transfer learning for nlp, in: International conference on machine learning, PMLR. pp. 2790–2799.

[12] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al., 2022. Lora: Low-rank adaptation of large language models. ICLR 1, 3.

[13] Iliakopoulou, N., Stojkovic, J., Alverti, C., Xu, T., Franke, H., Torrellas, J., 2025. Chameleon: Adaptive Caching and Scheduling for Many-Adapter LLM Inference Environments. Association for Computing Machinery, New York, NY, USA. p. 217–231. URL: https://doi.org/10.1145/3725843.3756083.

[14] Jaiswal, S., Arun, S., Parayil, A., Mallick, A., Mastorakis, S., Khare, A., Alverti, C., Amant, R.S., Bansal, C., Rühle, V., et al., 2025. Serving heterogeneous lora adapters in distributed llm inference systems. arXiv preprint arXiv:2511.22880 .

[15] Johnson, D.S., 1974. Fast algorithms for bin packing. Journal of Computer and System Sciences 8, 272–314. URL: https://www.sciencedirect.com/science/article/pii/S0022000074800267, doi:https://doi.org/10.1016/S0022-0000(74)80026-7.

[16] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J., Zhang, H., Stoica, I., 2023. Efficient memory management for large language model serving with pagedattention, in: Proceedings of the 29th Symposium on Operating Systems Principles, pp. 611–626.

[17] Lam, S.K., Pitrou, A., Seibert, S., 2015. Numba: A llvm-based python jit compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, pp. 1–6.

[18] Li, S., Lu, H., Wu, T., Yu, M., Weng, Q., Chen, X., Shan, Y., Yuan, B., Wang, W., 2025. Toppings: Cpu-assisted, rank-aware adapter serving for llm inference, in: 2025 USENIX Annual Technical Conference (USENIX ATC 25), pp. 613–629.

[19] Li, X.L., Liang, P., 2021. Prefix-tuning: Optimizing continuous prompts for generation. Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers) , 4582–4597URL: https://api.semanticscholar.org/CorpusID:230433941.

[20] Liu, H., Tam, D., Muqeeth, M., Mohta, J., Huang, T., Bansal, M., Raffel, C.A., 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. Advances in Neural Information Processing Systems 35, 1950–1965.

[21] López, F.A., Oliveras, J., Wang, C., Gutierrez-Torre, A., Tardieu, O., Youssef, A., Torres, J., Berral, J.L., 2025. A data-driven ml approach for maximizing performance in llm-adapter serving, in: 9th Machine Learning for Systems (ML for Systems) Workshop, NeurIPS 2025. Poster session, San Diego, CA.

[22] Microsoft, 2022–2025. DeepSpeed-MII. GitHub repository. URL: https://github.com/deepspeedai/DeepSpeed-MII.

[23] NVIDIA, 2023–2025. TensorRT-LLM. GitHub repository. URL: https://github.com/NVIDIA/TensorRT-LLM.

[24] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830.

[25] Qin, G., Eisner, J., 2021. Learning how to ask: Querying lms with mixtures of soft prompts, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 5203–5212.

[26] Qiu, H., Biswas, A., Zhao, Z., Mohan, J., Khare, A., Choukse,

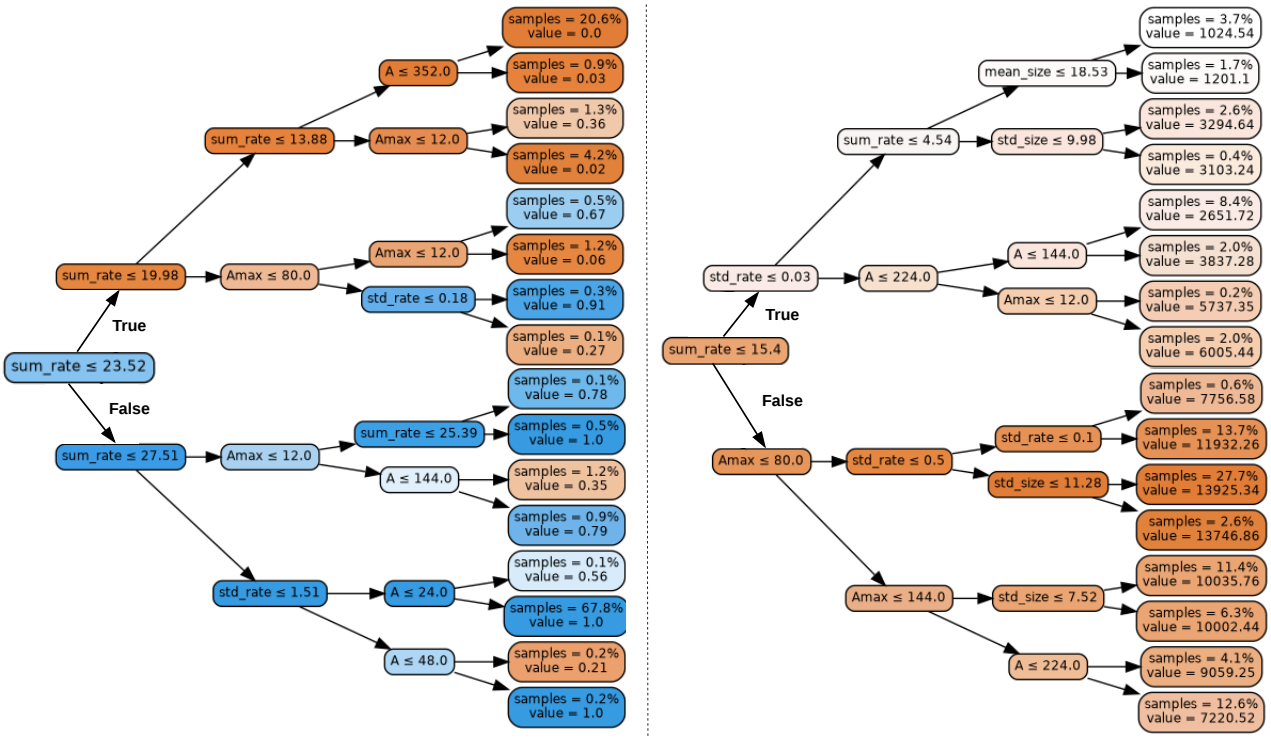


Figure C.14: Shallow decision trees derived from the best performing RF model to improve inference performance and enhance model interpretability. (Left) To estimate starvation for Llama-3.1-8B-Instruct. Final value represents the probability of starvation arising. (Right) To estimate throughput for Qwen2.5-7B-Instruct. Final value represents the expected throughput in toks/s.

E., Goiri, Í., Zhang, Z., Shen, H., Bansal, C., Ramjee, R., Fonseca, R., 2025. Modserve: Modality- and stage-aware resource disaggregation for scalable multimodal model serving, in: Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC 2025), Association for Computing Machinery, New York, NY, USA.

[27] Recasens, P.G., Agullo, F., Zhu, Y., Wang, C., Lee, E.K., Tardieu, O., Torres, J., Berral, J.L., 2025. Mind the memory gap: Unveiling gpu bottlenecks in large-batch llm inference, in: 2025 IEEE 18th International Conference on Cloud Computing (CLOUD), IEEE. pp. 277–287.

[28] Recasens, P.G., Zhu, Y., Wang, C., Lee, E.K., Tardieu, O., Youssef, A., Torres, J., Berral, J.L., 2024. Towards pareto optimal throughput in small language model serving, in: Proceedings of the 4th Workshop on Machine Learning and Systems, pp. 144–152.

[29] Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., Sundaram, R., 2019. Nexus: a gpu cluster engine for accelerating dnn-based video analysis, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, USA. p. 322–337. doi:10.1145/3341301.3359658.

[30] Shen, Z., He, Y., Wang, Z., Zhang, Y., Sun, G., Ye, W., Li, A., 2025. EdgeLoRA: An Efficient Multi-Tenant LLM Serving System on Edge Devices. Association for Computing Machinery, New York, NY, USA. p. 138–153. URL: https://doi.org/10.1145/3711875.3729141.

[31] Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., et al., 2024. Slora: Scalable serving of thousands of lora adapters. Proceedings of Machine Learning and Systems 6, 296–311.

[32] Sung, Y.L., Cho, J., Bansal, M., 2022. Lst: Ladder side-tuning for parameter and memory efficient transfer learning. Advances in Neural Information Processing Systems 35, 12991–13005.

[33] Touvron, H., et al., 2023. Llama 2: Open foundation and fine-tuned chat models. URL: https://arxiv.org/abs/2307.09288, arXiv:2307.09288.

[34] Virtanen, P., et al., 2020. Scipy 1.0: Fundamental algorithms for scientific computing in python. Nature Methods 17, 261–272. doi:10.1038/s41592-019-0686-2.

[35] Wang, L., Chen, S., Jiang, L., Pan, S., Cai, R., Yang, S., Yang, F., 2025. Parameter-efficient fine-tuning in large language models: a survey of methodologies. Artificial Intelligence Review 58, 227. URL: https://doi.org/10.1007/s10462-025-11236-4, doi:10.1007/s10462-025-11236-4.

[36] Wengwengwhale, 2024. Finance lora adapter for llama-3.1-8b instruct. URL: https://huggingface.co/Wengwengwhale/llama_3.1_8B_Instruct_Finance_lora_adapter.

[37] Wu, B., Zhu, R., Zhang, Z., Sun, P., Liu, X., Jin, X., 2024. dlora: Dynamically orchestrating requests and adapters for lora llm serving, in: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pp. 911–927.

[38] Yang, A., et al., 2025. Qwen2.5 technical report. URL: https://arxiv.org/abs/2412.15115, arXiv:2412.15115.

[39] yard1, 2024. Sql lora for llama-2-7b. URL: https://huggingface.co/yard1/llama-2-7b-sql-lora-test.

[40] Yu, G.I., Jeong, J.S., Kim, G.W., Kim, S., Chun, B.G., 2022. Orca: A distributed serving system for transformer-based generative models, in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pp. 521–538.

[41] Zhang, H., Tang, Y., Khandelwal, A., Stoica, I., 2023. Shepherd: Serving dnns in the wild, in: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pp. 787–808.

[42] zjudai, 2025. Medical lora for qwen2.5-7b-instruc. URL: https://huggingface.co/zjudai/flowertune-medical-lora-qwen2.5-7b-instruct.