xLoRA: Faster and Cheaper LoRA LLM Serving with Serverless Computing

Abstract

Serverless computing has grown rapidly for serving Large Language Model (LLM) inference due to its pay-as-you-go pricing, fine-grained GPU usage, and rapid scaling. However, our analysis reveals that current serverless platforms can effectively serve general LLM but fail to efficiently handle Low-Rank Adaptation (LoRA) inference due to three key limitations: 1) massive parameter redundancy among functions where 99% of weights are unnecessarily duplicated, 2) costly artifact loading latency beyond LLM loading, and 3) magnified resource contention when serving multiple LoRA LLMs. These inefficiencies lead to massive GPU wastage, increased Time-To-First-Token (TTFT), and high monetary costs.

We propose *xLoRA*, a novel serverless inference system designed for faster and cheaper LoRA LLM serving. *xLoRA* enables secure backbone LLM sharing across isolated LoRA functions to reduce redundancy. We design a pre-loading method that pre-loads comprehensive LoRA artifacts to minimize cold-start latency. Furthermore, *xLoRA* employs contention-aware batching and offloading to mitigate GPU resource conflicts during bursty workloads. Experiments on industrial workloads demonstrate that *xLoRA* reduces TTFT by up to 86% and cuts monetary costs by up to 89% compared to state-of-the-art LLM inference solutions.

1 Introduction

large language models (LLMs) have rapidly become the computation engine behind AI products. Two complementary patterns now dominate LLM inference. The first involves directly using pre-trained models such as Llama [55] and Qwen [44], while the second fine-tunes the pre-trained LLM for specific domains or tasks. In this scenario, Low-Rank Adaptation (LoRA) has emerged as the dominant fine-tuning technique due to its efficiency, enabling users to inject all task-specific knowledge into a lightweight adapter without retraining the full model.

Serving LLMs at scale is challenging due to stringent response time requirements and high GPU costs. Users expect sub-second response time for the first token, while even a 7B LLM already saturates an entire NVIDIA A10 during inference. Furthermore, providers face the complex task of hosting numerous LLM versions to accommodate a wide range of users and applications. For example, major cloud services are required to maintain a catalog of foundational LLMs alongside thousands of fine-tuned variations to meet diverse customer needs [8, 49]. Maintaining numerous models on long-running GPU instances results in substantial resource and monetary costs.

To alleviate these issues, serverless inference has emerged as a promising paradigm. Serverless platforms offer pay-as-you-go pricing, fine-grained GPU usage, and flexible architectures to support multiple LLMs with rapid scaling capabilities to handle varying workloads. Many serverless LLM inference solutions leverage these benefits, including Amazon Bedrock [8], NVIDIA DGX [15], and ServerlessLLM [20].

However, we observed that existing serverless solutions fail to effectively serve LoRA-based LLMs due to overlooking LoRA's unique characteristics.

Observation 1: Backbone redundancy among LoRAs. Mainstream serverless platforms isolate each function with independent execution environments and CUDA contexts, preventing cross-function memory sharing [7, 38]. This forces identical backbone LLM[1] weights to be repeatedly loaded per function instance, even though LoRA adapters modify less than 1% of parameters [11, 58], making over 99% of loaded weights redundant across LoRA functions.

Observation 2: LoRA introduces costly artifacts[2] loading latency. While existing serverless LLM solutions [20, 33, 62] have focused on accelerating the loading of the backbone checkpoint (from RAM, SSD, or remote storage), they overlook the substantial overhead from other essential artifacts. As we demonstrate in Fig. 1, the time required to load libraries, LoRA adapters, and just-in-time (JIT) compiled CUDA ker-

[1] We use "backbone" to refer to the base LLM.

[2] In this paper, an artifact is any component that must be loaded for the function to execute, including the LoRA adapter, dependent software libraries, and just-in-time (JIT) compiled CUDA kernels.

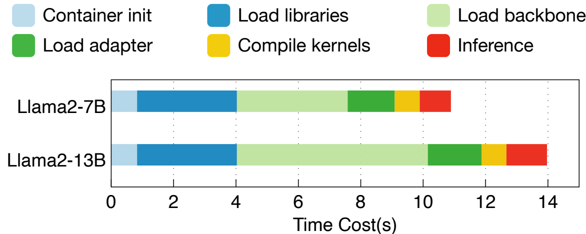


Figure 1: Time breakdown of a LoRA LLM inference with serverless computing.

nels can actually exceed the latency of loading the backbone checkpoint itself. For generating 100 tokens on NVIDIA L40S GPU, the overall loading latency exceeds LLM inference by over 10×.

Observation 3: Serving multi-LoRA magnifies resource contention. Serverless isolation forces each function to run in an independent CUDA context, requiring separate fetches of model parameters from GPU global memory to Streaming Multiprocessors (SM). In the common scenario where multiple LoRA functions execute concurrently [36, 66], these parallel transfers saturate the GPU's internal memory bus, causing contention that dramatically increases Time-To-First-Token (TTFT) and violates service level objective (SLO).

These observations motivate our goal: a *cheaper*, *faster*, and *more flexible* serverless LoRA inference system. Concretely, we aim to 1) enable LoRA functions to share the 99%-dominant backbone in GPU memory to cut resource and monetary costs, 2) minimize cold-start latency by pre-loading LoRA artifacts, and 3) support multiple backbones and LoRA adapters simultaneously.

However, achieving these goals introduces fundamental challenges: *First*, the strict isolation of serverless functions hinders the secure and efficient sharing of a single backbone model across their boundaries—a challenge that current backbone sharing solutions [11, 49, 58] cannot address. *Second*, although pre-loading is critical for achieving low latency, its high memory demand limits GPU availability for maximizing pre-loading acceleration. *Third*, concurrent functions on a single GPU compete for shared resources, creating a critical trade-off: low concurrency under-utilizes the GPU, while high concurrency causes interference.

To address these challenges, we present *xLoRA*, a cost-efficient serverless inference system designed for LoRA LLM serving. First, *xLoRA* introduces a decoupled architecture that places the backbone in a secure, shared environment. It allows multiple isolated LoRA functions to access it, thus preserving security while enabling massive memory savings. Second, *xLoRA* employs an opportunistic pre-loading policy that strategically pre-loads artifacts to minimize cold-start latency. By pre-loading these artifacts on idle instances already provisioned by the serverless provider, it maximizes acceleration without extra memory overheads. Third, *xLoRA* implements contention-aware batching to maximize GPU utilization while managing interference. This dual strategy increases batch sizes within functions for efficiency while limiting concurrent function execution to prevent contention and ensure SLO compliance.

We summarize *xLoRA*'s contributions as follows:

- We observed that serverless LoRA inference suffers from exclusive GPU occupation and excessive cold-start latency, leading to significantly high monetary costs and increased TTFT.
- We design *xLoRA*, a novel serverless framework that makes LoRA inference faster and cheaper. Leveraging backbone sharing and extended pre-loading of LoRA artifacts, *xLoRA* substantially reduces GPU consumption and cold-start latency.
- We thoroughly evaluate *xLoRA* on industrial workloads and benchmark LLMs. Extensive experiments show reductions of up to 86% TTFT and 89% in monetary costs, compared to state-of-the-art solutions.

2 Background and Motivation

2.1 Serverless for LLM Inference

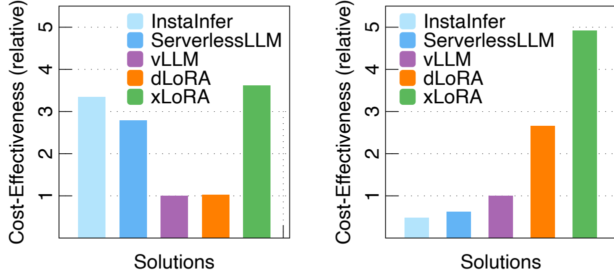
Serverless computing is an increasingly compelling paradigm for deploying LLM inference services, and its effectiveness has been demonstrated by major cloud providers like AWS and NVIDIA who use it to serve LLMs at scale [8, 15]. The paradigm is function-centric: user code is executed in independent, sandboxed processes (e.g., containers or microVMs [2]) to ensure security and isolation in a multi-tenant environment. When applied to LLM inference, the lifecycle begins with a significant initialization phase. The system must first provision a container, load libraries like PyTorch, transfer the large model parameters to GPU memory, and compile the necessary CUDA kernels. Only after this entire "cold-start" sequence is complete can the function begin processing inference requests.

This serverless model offers significant advantages in efficiency and elasticity over traditional serverful[3] deployments such as vLLM [28]:

First, it provides superior cost and GPU efficiency. The pay-per-use model eliminates charges for idle GPUs during periods of low traffic, which is common for workloads with variable request arrivals. We adopt the cost-effectiveness metric from existing studies [26, 54, 56], defined as $1/(\text{E2E_latency} \times \text{Cost})$, to jointly consider both latency and monetary cost. Our evaluation in Fig.2a shows that when serving a Llama2-7B model, serverless solutions [20, 53] are 3× more cost-effective than serverful counterparts [28, 58].

Second, serverless delivers superior elasticity. LLM inference workloads often face dramatic load fluctuations, with traffic peaks reaching 34.6× higher than valleys [52]. Unlike serverful systems that typically scale on a minute-level,

[3] We use "serverful" to refer to VM-based long-running serving.



(a) **Cost-effectiveness of serverless and serverful solutions for one Llama2-7B base LLM.** (b) **Cost-effectiveness of serverless and serverful solutions for four Llama2-7B LoRA LLMs.**

Figure 2: Cost-effectiveness of serverless and serverful solutions (we set vLLM as baseline).

serverless platforms can scale within seconds to absorb sudden bursts, thereby ensuring consistent performance under highly dynamic conditions.

2.2 Low-Rank Adaptation (LoRA)

While serverless computing offers advantages for general LLM inference, real-world applications rarely rely on general-purpose models. Organizations typically deploy multiple specialized models fine-tuned for specific domains, tasks, or user segments. LoRA has become the predominant fine-tuning method due to its parameter efficiency, allowing practitioners to create specialized models without retraining the entire backbone LLM. Based on the observation that weight updates have low intrinsic dimensionality, LoRA freezes pre-trained weights (backbone model) and injects trainable low-rank matrices, reducing parameters by factors of 10,000 while maintaining performance.

For a pre-trained weight matrix $W \in \mathbb{R}^{h \times d}$, LoRA introduces matrices $A \in \mathbb{R}^{h \times r}$ and $B \in \mathbb{R}^{r \times d}$ where $r \ll \min(h,d)$. The updated weight becomes $W' = W + AB$, transforming the forward pass from $h = xW$ to:

$$h = x(W + AB) = xW + xAB,$$

which combines the frozen base model output (xW) with the trainable adapter output (xAB). Due to the decoupling of backbone and LoRA parameters, the inference can be losslessly operated by separately calculating the attention of the backbone and LoRA adapter, and finally gathering their results as the final output.

2.3 Serverless' Limitation for LoRA Inference

Massive GPU wastage due to backbone redundancy. Our observations reveal that existing serverless platforms require each function to load the entire backbone LLM independently, despite sharing the same underlying model. This redundancy manifests in excessive GPU consumption and elevated monetary costs. Since even a 7B parameter LLM consumes an

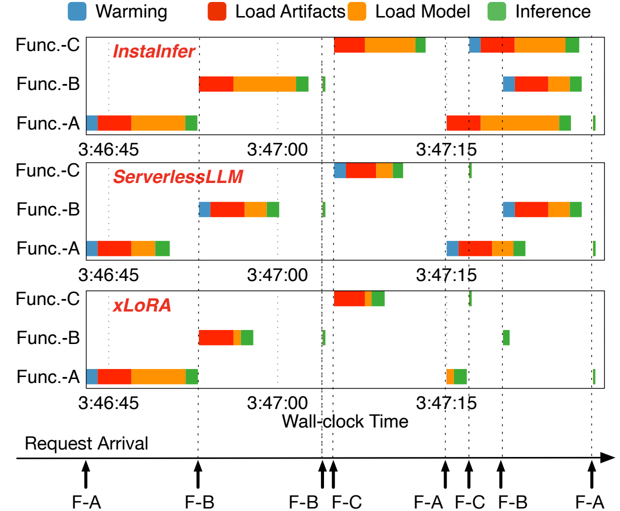


Figure 3: LoRA inference function invocations' time breakdown.

entire NVIDIA A10 GPU during inference, and the backbone accounts for up to 99% of an LLM's parameters, this results in substantial wasted GPU resources. The problem is exacerbated by serverless platforms' keep-alive policies, which maintain each invoked function for several minutes after execution to mitigate cold-starts. In the LoRA scenario, each function's complete LLM occupies expensive GPU resources during idle periods, multiplying costs unnecessarily. Given that GPU costs constitute approximately 90% of an invocation's total monetary expense[4], this inefficient resource utilization translates directly to significantly higher operational costs. As Fig. 2b shows, when serving four LoRA functions fine-tuned on Llama2-7B LLM, serverless solutions' cost-effectiveness decreases significantly due to massive redundancy.

Heavy cold-start latency due to artifacts loading. Loading a backbone LLM with billions of parameters takes several seconds per function. Since each function independently loads the identical backbone, aggregate cold-start latency increases linearly with the number of LoRA functions. Furthermore, current serverless inference also overlooks initialization delays, including loading necessary libraries and LoRA adapters, and compiling CUDA kernels. These prolonged startup times prevent fast scaling and make it challenging to serve bursty workloads effectively.

We visualize E2E latency breakdown for three Llama2-13B LoRA functions using the Azure Trace in Fig. 3. Both InstaInfer and ServerlessLLM show high cold-start latency from repeated backbone and LoRA loading. Despite ServerlessLLM's improvements in backbone loading, cold-start latency remains substantial.

Loss of serverless isolation guarantees due to backbone sharing. Solutions like Punica [11], S-LoRA [49], and dLoRA [58] address efficiency by sharing a single backbone

[4]Based on the Alibaba Cloud serverless pricing model [13].

but demolish the isolation that defines the serverless paradigm. Their single-process architecture breaks serverless's multi-tenancy guarantees: Its globally shared data breaks security isolation, creating vulnerabilities for sensitive data to leak between tenants. It also breaks fault isolation, as a single mis-behaving tenant can crash the entire system. Finally, it breaks performance isolation by introducing the "noisy neighbor" problem, where one user degrades service for all others.

2.4 Motivating Backbone Sharing and Pre-loading

Our analysis reveals two key insights. *First*, significant cost and latency arise from deploying a full LLM model for each LoRA function, even when they share the same backbone. This motivates our first approach: enabling backbone sharing across multiple function instances to reduce GPU usage, deployment cost, and model loading time. When serving eight Llama-13B functions, backbone sharing can reduce up to 86% GPU usage and 79% monetary cost while reducing 42% cold-start latency. *Second*, we note that loading LLM artifacts accounts for over 90% of the startup time. To mitigate this bottleneck, our approach is to pre-load these artifacts prior to function invocation, which reduces startup latency and improves throughput under bursty workloads. With the artifacts already loaded when a request arrives, the system can proceed directly to inference, accelerating the TTFT by over 16.8×.

2.5 Opportunity of Backbone Sharing and Pre-Loading

Opportunity for backbone sharing: Backbone sharing is both feasible and beneficial for two reasons. First, the number of LoRA-adapted models significantly exceeds backbone LLMs in practical deployments. For example, the Llama2 family has generated 11,258 LoRA adapters, indicating many functions can share the same backbone. Second, the inference calculation for the backbone and LoRA adapter can be performed separately and added up together, ensuring that the sharing does not compromise the inference accuracy of individual functions. This unmerged calculation of backbone and adapter does not affect any inference result and has been applied in current solutions [11, 49, 58].

Opportunity for pre-loading: In current serverless platforms, maintaining idle instances (including container and GPU) in advance for serving future requests is common to mitigate cold-starts [10, 21, 32, 39, 46, 48, 51]. To deal with bursty workload, functions are allocated with enough memory to serve the peak workload (large concurrency). Thus, there is a huge gap between a function instance's idle and busy state [22, 45, 48, 61, 65, 68]. We observe that loading all artifacts (excluding the backbone) accounts for only 20% of the allocated memory. This large memory gap between the running and idle states provides a "free-lunch" opportunity: functions' artifacts can be pre-loaded into existing idle instances created by the platform, rather than requiring the creation of new instances.

3 *xLoRA*: A Bird's-Eye View

3.1 Goals & Challenges

xLoRA aims to achieve three goals:

Minimal TTFT Latency. Reduce startup latency from both container initialization and LLM artifact loading.

Minimal Resource and Monetary Cost. As serverless providers charge based on both resource usage and execution time, *xLoRA* should minimize both to lower overall cost.

High Scalability. Under bursty workloads, *xLoRA* should launch new functions promptly without violating SLOs.

To meet our goals, we must address three tough challenges:

How to reduce the function startup latency while guaranteeing isolation and security? To satisfy the isolation standard of serverless, each function should run in an independent process, operating inference using its own computing resource and managing KV cache and other intermediate data in its own memory stack. It's challenging to share the backbone LLM under strict isolation requirements.

How to achieve maximal acceleration performance with limited GPU resources? The backbone LLM, adapters, and user libraries all consume significant memory. To optimize performance, we must carefully decide which components of which functions should be pre-loaded into the high-value GPU memory, and which can reside in the less valuable container memory.

How to achieve fast scale-up under bursty workloads? As the initialization of LLM functions is heavy, during bursty workloads, we should offload unrelated pre-loaded artifacts from GPUs to make room for future requests. To further accelerate function initialization, function instances should reside on GPUs that have already loaded corresponding backbone LLMs for locality awareness.

3.2 *xLoRA*'s System Architecture

We introduce the architecture of *xLoRA*, a model-sharing based serverless LLM inference system aiming to minimize startup latency and achieve high scalability under minimal resource and monetary cost. We design a secure LLM sharing mechanism that allows functions to share the backbone LLM safely. To ensure the model sharing mechanism can achieve the above goals, *xLoRA* contains four components: The Pre-Loading Scheduler, the Batching Scheduler, the Pre-Loading Agent, and the Dynamic Offloader.

Pre-Loading Scheduler determines which artifacts of which function should be pre-loaded in each container and GPU. Specifically, a function's libraries should be pre-loaded

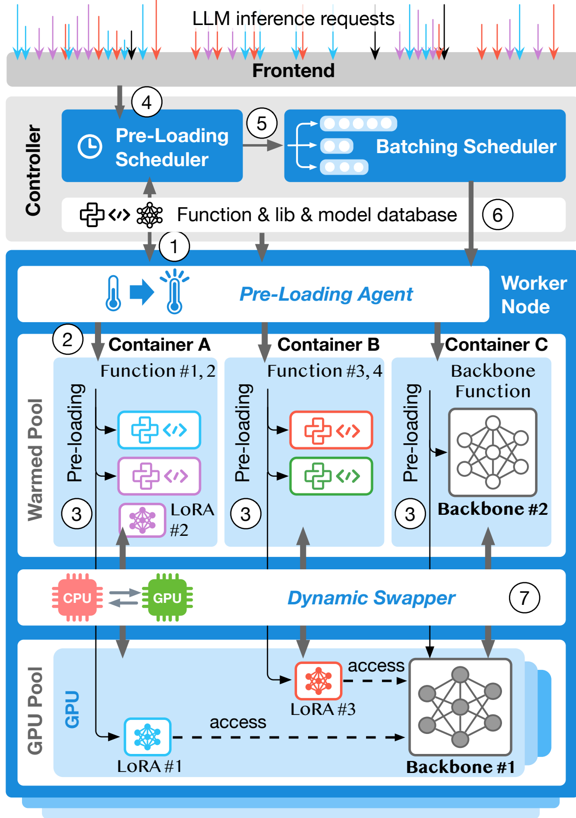


Figure 4: System overview.

in container memory, the CUDA runtime and kernels must be pre-loaded in GPU memory, and the model can be pre-loaded in either. Therefore, it's essential to determine the optimal pre-loading decision under limited resources.

Batching Scheduler determines the batch size and queuing time of each function. It aims at fully utilizing GPU resources to maximize throughput under the function's SLO.

The Pre-Loading Agent runs in each worker node. It receives the pre-loading decision from the Pre-Loading Scheduler and sends the corresponding command to each container and GPU. Furthermore, it manages the access of LoRA functions to the backbone.

The Dynamic Offloader detects whether a GPU has enough remaining space for serving the arriving requests. When bursty requests arrive, it offloads unrelated functions' artifacts to container memory or totally removes them, until there is enough space to serve all requests.

3.3 System Workflow

Pre-loading Decision and Execution (Steps 1-3): The Pre-Loading Scheduler analyzes resource availability and function request frequency to make optimal loading decisions. This is a complex task where the scheduler must decide whether to fully load a function into GPU memory for minimal latency or place it in container memory for balanced performance. Following this decision, the system loads the required libraries, models, and CUDA kernels into the designated memory locations.

Request Batching (Steps 4-6): When a request arrives, the scheduler selects the most suitable function instance based on its pre-loaded components. Concurrently, a Batching Scheduler gathers incoming requests for each function. Managing these batches to optimize throughput by balancing batch size against latency is a significant challenge. Once the batch size is reached or the batching timeout occurs, requests are sent to the chosen instance.

Dynamic GPU Memory Management (Step 7): Running in parallel to request handling is the GPU memory management. The Dynamic Offloader constantly monitors the GPU's available capacity. As the GPU memory fills, the Offloader must intelligently identify and move the models of non-invoked functions to container memory or clear their CUDA contexts, freeing up space for active requests without degrading pre-loading hit rate.

4 *xLoRA*'s Design

4.1 LLM Artifacts Pre-loading

To achieve maximum acceleration performance with minimal resource wastage, we design two principles for pre-loading: 1) LLM artifacts are only pre-loaded in existing idle container and GPU instances. We never proactively create new instances for pre-loading. 2) To deal with peak workload, serverless functions are usually over-allocated with more resources than pre-loading its own artifacts [18, 22, 45, 48, 61, 65, 68]. Thus, we share the container among multiple functions in the pre-loading stage.

xLoRA manages four types of LLM artifacts: libraries, backbone models, LoRA adapters, and CUDA kernels. Since these artifacts must be loaded in sequence (e.g., loading CUDA kernels requires libraries and container components), we approach pre-loading as a Precedence-Constrained Knapsack Problem (PCKP).

In our formulation, we aim to pre-load serverless functions (F) within idle containers (C) and GPUs (G). Each function's each artifact (i) specific memory requirements(w_i^f) and offers potential pre-loading benefit(v_i^f), which is calculated as the product of loading delay and the predicted request arrival rate. Our objective is to maximize the cumulative performance benefit while respecting memory constraints.

As the container and GPU cannot hold all artifacts, our optimization objective is to maximize the cumulative performance benefit derived from pre-loading LLM artifacts. Since libraries can only be pre-loaded on containers, CUDA kernels on GPUs, and backbones and adapters on both, we comprehensively consider the benefits of pre-loading artifacts on container and GPU. We formulate this as:

$$\max \sum_{f \in F} \sum_{i \in A_f} \left[\sum_{c \in C} v_i^f x_i^{fc} + \sum_{g \in G} v_i^f x_i^{fg} \right], \tag{1}$$

where x is a binary that represents whether this artifact is pre-loaded.

The pre-loading decisions are guided by three constraints: 1) *Capacity constraints*: limit memory usage in containers and GPUs. 2) *Loading order precedence constraints*: libraries before models, models before CUDA kernels. 3) *Backbone-adapter coupling constraints*: adapter and its backbone LLM must be placed on the same GPU. As PCKP is NP-hard [42], finding optimal solutions requires exponential time, making it impractical for serverless environments requiring millisecond-scale scheduling.

Therefore, we employ a greedy heuristic that sorts artifacts by their value density $\rho_i^f = v_i^f / w_i^f$ and iteratively pre-loads them while respecting constraints. Formally, at each step k, we select the best combination of artifact i, function f, and target container & GPU t:

$$(i^*, f^*, t^*) = \arg \max_{(i,f,t) \in \mathcal{F}_k} \rho_i^f \tag{2}$$

where $\mathcal{F}_k$ denotes the set of feasible artifact-location pairs at step k that satisfy all constraints.

Our greedy approach reduces the complexity from exponential $O(2^{F(|C|+|G|)})$ to polynomial $O(|F|^2 \cdot (|C|+|G|))$, making it practical for large-scale deployments. Our evaluation in Sec. 5.12 demonstrates that it delivers near-optimal pre-loading decisions while meeting serverless latency requirements.

4.2 Adaptive Batching

To improve throughput and reduce cold-starts, *xLoRA* batches requests into pre-loaded function instances, allowing multiple requests to benefit from the same LLM artifacts and further reducing cold-starts. However, serverless environments introduce a critical challenge: each function runs in an isolated CUDA context. Consequently, each must independently fetch model parameters from the GPU's global memory to SM for computation. When multiple functions run concurrently, these redundant data transfers saturate the GPU's internal memory bus, creating a severe I/O bottleneck. This causes destructive interference, where each additional concurrent function significantly slows down all others.

To address this, *xLoRA* adopts a dual strategy: maximize batch sizes within functions to amortize loading costs, while minimizing concurrent function execution to prevent I/O contention. Large intra-function batches improve GPU utilization, but excessive inter-function concurrency overwhelms memory bandwidth and violates SLOs. We formulate a two-layer batching approach that balances high per-function throughput with system-wide I/O constraint management to ensure SLO compliance.

At the local level, each function i implements fill-or-expire batching. Due to the computationally intensive pre-filling stage, TTFT increases linearly with batch size:

$$T_i(b) = T_{0,i} + \alpha_i(b-1), \tag{3}$$

where $T_{0,i}$ is the base inference time for the first token and α_i is the marginal cost per additional request. Through offline profiling, we determine the maximum batch size B_i within the SLO. The system calculates maximum batch delay based on current batch number N_i:

$$d_i = SLO_i - T_i(N_i). \tag{4}$$

Batching stops when either N_i requests are collected or delay d_i expires. This design ensures SLO compliance while allowing longer wait times for smaller batches to better utilize pre-loaded artifacts.

At the global level, the Batching Scheduler manages resource contention when multiple batches compete for GPU resources. With M concurrent batches on a shared GPU, inference time expands to:

$$T_i^{\text{eff}}(b) = M \cdot T_i(b). \tag{5}$$

The scheduler prioritizes batches by their *deadline margin*:

$$\Delta_i = SLO_i - (w_i + M \cdot T_i(b)), \tag{6}$$

where w_i is time already spent waiting. Batches with smaller margins receive priority, while those with larger margins can wait longer to accumulate more requests.

4.3 Dynamic GPU Offloading

Adaptive batching enables each function instance to handle up to a maximum batch size while maintaining minimal TTFT. However, reaching this capacity requires sufficient GPU memory for KV caches.

Pre-loaded artifacts consume significant GPU memory even when their functions are idle. During burst periods, when functions must serve many concurrent requests, these unused artifacts limit available memory for KV caches, preventing functions from reaching maximum batch size.

Therefore, we propose the Dynamic Offloader to remove unrelated LLM artifacts from GPUs, ensuring invoked functions have sufficient memory for maximum batching.

Offloading Policy Design. Our policy minimizes performance degradation by removing the least valuable artifacts. We formulate this as a value-optimization problem. When GPU g requires Q_g additional memory:

$$\sum_{i \in I_g} w_i \cdot x_i \geq Q_g \tag{7}$$

The objective minimizes performance loss:

$$\min \sum_{i \in I_g} v_i \cdot x_i \tag{8}$$

where $x_i \in 0,1$ indicates whether to offload artifact i, v_i represents its performance value, w_i its memory footprint, and I_g the artifacts on GPU g. Each function has two artifacts: adapter model and CUDA contexts.

Since this problem is NP-hard and shares similar constraints with pre-loading, we employ the same greedy algorithm based on artifact value density. This policy executes in microseconds, enabling rapid offloading decisions.

4.4 Backbone LLM Sharing

Existing methods for sharing a backbone LLM [11, 49, 58] are incompatible with the serverless paradigm, which requires strict process and resource isolation for security and billing. These methods bundle the backbone and LoRA adapters into a single process, which serverless architecture forbids.

To solve this, our approach decouples the static backbone model from dynamic inference components. As Fig. 5 shows, a dedicated backbone function loads the LLM's tensors onto a GPU and exposes them via CUDA Inter-Process Communication (IPC) handles. This allows multiple, independent LoRA functions to access the shared backbone tensors with zero memory duplication. Each LoRA function then initializes an empty, structure-only backbone model and maps its tensors to this shared memory region using the IPC handles. This gives the runtime the necessary model structure and CUDA computing graph without duplicating the backbone weights.

With this design, each function executes inference computations independently using its own allocated resources, preserving the isolation and security guarantees of serverless computing. This memory-efficient approach allows a single GPU to host hundreds of LoRA functions simultaneously. To ensure the shared backbone remains read-only, the system calculates backbone and adapter outputs separately and combines the results. Finally, a fault-tolerance mechanism ensures continuity. If a backbone function fails, the system automatically redirects all dependent LoRA functions to a healthy backup, minimizing downtime.

4.5 Security and Privacy

Backbone sharing security guarantee. To secure the shared backbone from unauthorized access and weight tampering across untrusted functions, we implement two mechanisms. *1) Ephemeral IPC handler transfer.* We transfer CUDA IPC handles via ephemeral Unix domain sockets using `SCM_RIGHTS`, not persistent files. After a one-shot transfer, the server closes the descriptor and connection, blocking replays and enumeration. File-system permissions on the socket path gate access,

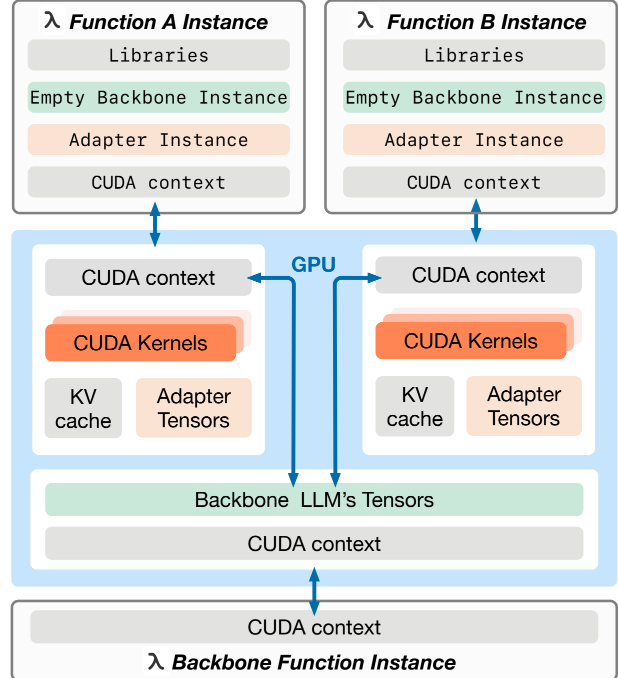


Figure 5: Backbone LLM sharing among functions.

and this capability-passing design is proven in secure file-transfer systems [1, 47, 57]. *2) Read-only memory protection.* We enforce hardware read-only access to shared tensors via CUDA virtual memory. The server maps with `PROT_READ`; clients receive driver-enforced read-only mappings. This preserves model-weight integrity and is validated by prior secure GPU data-sharing systems [24, 34].

Data isolation for pre-loaded functions. To ensure data isolation for functions within the same container, each function operates within isolated Linux namespaces with strict privilege controls and chroot jails, preventing unauthorized access to other processes' memory spaces or file systems through kernel-level enforcement.

Blackbox processing for data privacy. *xLoRA* is designed to treat user models and data as black boxes, ensuring it never accesses their plaintext content. We facilitate this by requiring a minor, two-line modification to the user's code, detailed in Appendix A.2. The *xLoRA* library automatically converts the adapter model into the necessary format and connects it to the backbone function.

5 Evaluation

We prototype *xLoRA* using 5.5K lines of Python code and 600 lines of CUDA code. Implementation details are available in Appendix A.1.

5.1 Experimental Setup

Testbed. Our experiments are conducted on two testbeds. The first is a single-node AWS EC2 GPU `g6e.48xlarge` instance with 384 CPU cores, 1,536 GB memory, and eight NVIDIA L40S GPUs. The second is a multi-node cluster on four AWS `g6e.24xlarge` instances, with a total of 768 CPU cores, 3,072 GB memory and 16 NVIDIA L40S GPUs.

Workload. To approximate real-world invocation patterns, we utilize serverless production traces from Azure Functions [48] and Azure LLM inference [40]. Despite being collected from serverless workloads, Azure Function traces are commonly used to represent LLM inference traffic in LLM serving studies [31, 58]. We categorize Azure Functions traces into three patterns based on the co-variance (CoV) of the request's inter-arrival time: "Predictable" (CoV ≤ 1), "Normal" ($1 < \text{CoV} \leq 4$), and "Bursty" (CoV > 4). Fig. 6 illustrates partial traces of the three patterns. From each pattern, eight 4-hour traces are randomly selected and mapped to individual functions. For the Azure LLM inference trace, as the function number is larger than that of LLMs, we map the trace to functions in a round-robin manner.

Models and machine learning (ML) Libraries. We select three backbone LLMs: Llama2-7B, Llama2-13B, and Llama2-70B. Each backbone is augmented with four popular LoRA adapters selected according to download trends on HuggingFace [19]. All inference pipelines are developed using PyTorch and Transformers. We evaluate Llama2-70B, which requires multiple GPUs, separately in Sec. 5.6.

Datasets and baselines. We use GSM8K [14], a real-world LLM dataset of human-created problems, as the prompt for each request. Two latest serverless and two serverful ML inference serving solutions are selected as baselines: 1) **InstaInfer** [53], a serverless inference system addresses mitigating ML artifacts loading latency for small models by preloading. 2) **ServerlessLLM** [20], the state-of-the-art serverless LLM inference framework for minimizing LLM checkpoint loading delay. We also choose two serverful approaches: 3) **dLoRA** [58], which is designed for serving multiple LoRA adapters concurrently. 4) **vLLM** [28], an memory-efficient LLM serving system. For fairness, we equip all solutions with batching, and for solutions that do not support backbone sharing, we let them use merge inference to speed up generation.

Evaluation Metrics. *Cold-start latency*: The time period before inference, including both container initialization and LLM artifacts loading. *Time-To-First-Token (TTFT)*: The time of a function from being triggered to return the first token. *Time-Per-Output-Token (TPOT)*: The average interval between each generated token. *Monetary cost*: The total money spent on running the whole workload. *Cost-effectiveness*: To evaluate inference efficiency, we propose a cost-effectiveness metric: $1/(E2E_latency \times Monetary_Cost)$. This captures the trade-off between speed and cost since optimizing latency alone risks GPU over-provisioning, while minimizing cost increases cold-starts. *Throughput*: The number of output tokens and served requests per unit time. *Scalability*: The ability to maintain latency/cost efficiency as workload or resource scales. *Overhead*: The additional latency and resource cost introduced by *xLoRA*.

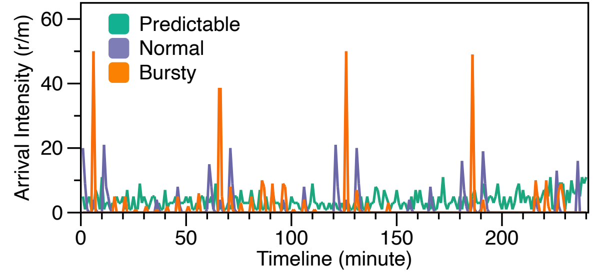


Figure 6: Trace example of "Predictable" (CoV ≤ 1), "Normal" ($1 < \text{CoV} \leq 4$), and "Bursty" request arrival pattern.

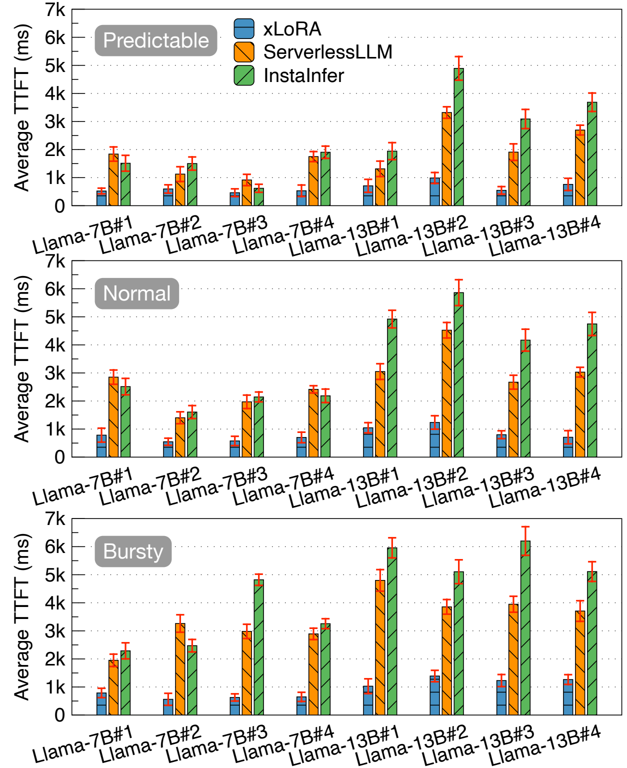


Figure 7: Average TTFT of the workloads at "Predictable", "Normal" , and "Bursty"arrival patterns.

5.2 TTFT and TPOT Evaluation

We evaluate the TTFT and TPOT of three serverless solutions on the 16-GPU cluster running four Llama2-7B based LoRA adapter functions and four Llama2-13B based LoRA adapter functions. The evaluation is conducted in three workloads: Predictable, Normal, and Bursty.

TTFT: Fig. 7 shows that *xLoRA* accelerates TTFT up to 4.7× and 7.1×, compared with ServerlessLLM and InstaInfer. As the result shows, for any type of LoRA function in any

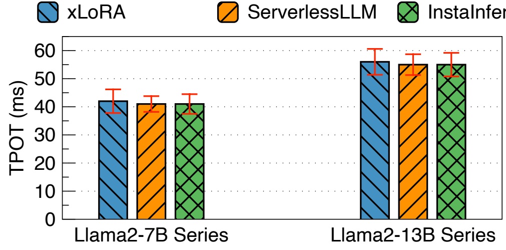


Figure 8: Average TPOT of *xLoRA* and baselines.

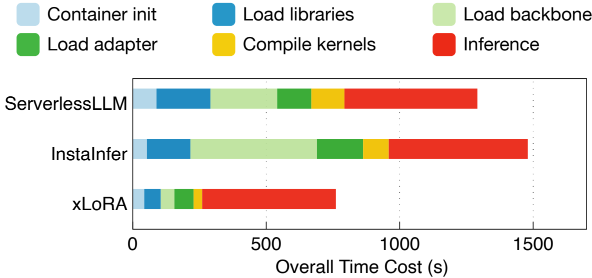


Figure 9: Time breakdown of the whole workload.

workload, *xLoRA* can significantly reduce its LLM artifact loading latency, thereby accelerating TTFT.

Although ServerlessLLM can accelerate LLM checkpoint loading from several seconds down to at least one second, it does not optimize the loading of libraries and CUDA kernels, nor can it effectively speed up the loading of LoRA adapters. InstaInfer, on the other hand, dynamically pre-loads and offloads function models and libraries for maximum acceleration. While this dynamic pre-loading performs well for small models, it is less effective for LLMs that require significantly more loading time. This frequent pre-loading and offloading substantially reduces the availability of function instances since they cannot begin inference during the pre-loading phase—explaining its poor performance with Llama2-13B series functions. Furthermore, all these solutions require loading both the backbone LLM and the LoRA adapter for every function. When a request arrives, they miss the opportunity to use another function's already-loaded backbone LLM to accelerate artifact loading. Additionally, they ignore the CUDA kernel compilation overhead during the first inference. These limitations further slow down TTFT.

TPOT: As workload patterns primarily affect cold-starts rather than inference execution, the TPOT measurements remain similar across Predictable, Normal, and Bursty workload scenarios. Fig. 8 shows, *xLoRA* does not significantly increase TPOT compared with other serverless solutions. Although *xLoRA*'s TPOT is 12% higher than that of baselines, it still remains within SLO requirements.

This slightly higher TPOT can be attributed to two main reasons: First, for the goal of minimizing TTFT and improving throughput, *xLoRA*'s Adaptive Batching Scheduler tends to collect more requests into a batch (while still adhering to SLO constraints). Second, *xLoRA* reduces replicas of back-

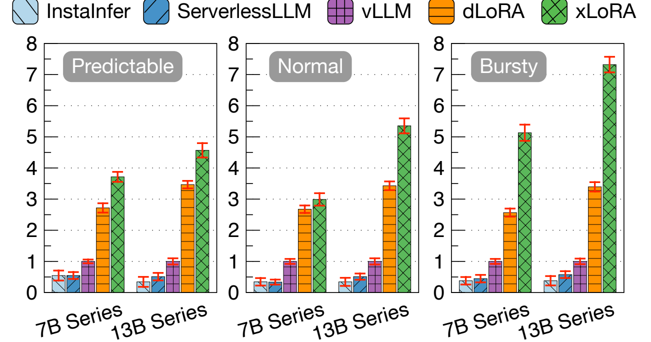


Figure 10: Cost-effectiveness of *xLoRA* and baselines running the Predictable, Normal, and Bursty workloads.

bone LLMs, which allows more GPU memory to be allocated for KV cache and enables larger maximum batch sizes (We further explain and evaluate this phenomenon in Sec. 5.7). As larger batch sizes require more computational resources, *xLoRA*'s average TPOT is moderately higher than that of InstaInfer and ServerlessLLM, both of which employ fixed and smaller batch sizes.

5.3 Time Breakdown Analysis

We further evaluate the cumulative time cost and breakdown of each solution running the "Normal" workload in Fig. 9. *xLoRA* significantly reduces the latency of loading each artifact, especially the backbone LLM. While InstaInfer and ServerlessLLM introduce higher cumulative cold-start latency than inference, indicating their limitations in LoRA serving.

5.4 Cost-Effectiveness

To evaluate *xLoRA*'s cost-effectiveness, we jointly consider E2E latency and monetary cost. We present relative cost-effectiveness with vLLM as baseline, using Alibaba Cloud serverless pricing [13]. Fig. 10 shows *xLoRA* outperforms all baselines across workloads. Against serverful systems (vLLM, dLoRA), *xLoRA* cuts costs severalfold with minimal cold-start overhead. Against serverless systems (InstaInfer, ServerlessLLM), *xLoRA* achieves up to 12.7× and 19.3× better performance through pre-loading and backbone sharing, reducing both latency and cost. Serverless baselines perform worse with Llama2-13B than Llama2-7B due to increased loading times and GPU demands, raising both latency and costs. InstaInfer, designed for million-parameter models like ResNet, shows the highest TTFT and cost when serving billion-parameter LLMs. This confirms that existing serverless solutions cannot directly serve LLMs. The detailed latency and cost is shown in Table. 1

5.5 Trade-off Between Latency and Cost

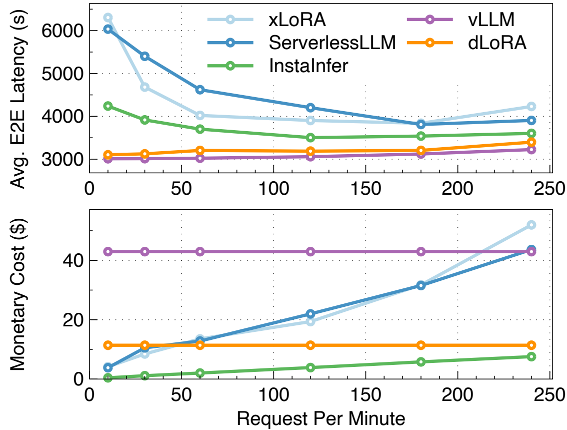


Figure 11: Effect of increasing workload intensity on E2E Latency and Cost.

Cost-effectiveness metrics equally weight latency and cost, which may not match real-world priorities. We instead evaluate latency and cost across varying workload intensities (scaling request rates with fixed arrival patterns), letting practitioners choose solutions based on latency or cost sensitivity.

Fig. 11 show a clear divide. Serverful solutions deliver consistently low latency but incur high, fixed costs that are inefficient during periods of low demand. Conversely, serverless baselines offer pay-per-use savings but suffer from severe latency, especially due to cold-starts at low traffic. *xLoRA* effectively bridges this gap. It achieves the cost-efficiency of a serverless architecture while maintaining performance nearly on par with serverful systems. Compared to dLoRA, the most efficient baseline, *xLoRA* achieves a 69% reduction in average operational costs with only a 17% increase in average latency. This trade-off shows that *xLoRA* provides significant cost savings with minimal performance impact, making it both practical and cost-effective.

5.6 Multi-GPUs and LLM Inference Traces

Current serverless platforms are typically evaluated with 7B and 13B models due to single-GPU limitation. To rigorously assess multi-GPU inference capabilities, we evaluate Llama2-70B with the Azure LLM inference trace. This larger model represents a production-grade service with real-world LLM inference patterns not captured by the sporadic Azure Function trace. Our evaluation was conducted on a server with four NVIDIA A100_80G GPUs serving four LoRA functions.

Compared to serverless baselines, the high memory requirements and loading latency of the 70B model significantly widen the performance gap between *xLoRA* and other serverless solutions. As demonstrated in Fig. 12, *xLoRA* maintains its performance advantages, achieving up to 3× acceleration while reducing up to 90% costs. Compared to serverful systems, which offer lower latency due to their long-running nature, this performance comes at a considerably higher monetary cost. *xLoRA* demonstrates significantly higher cost-effectiveness by reducing 91% cost with only 24% increase in latency. This result shows that *xLoRA*'s design effectively mitigates these expenses and can perform well under realistic LLM inference traces.

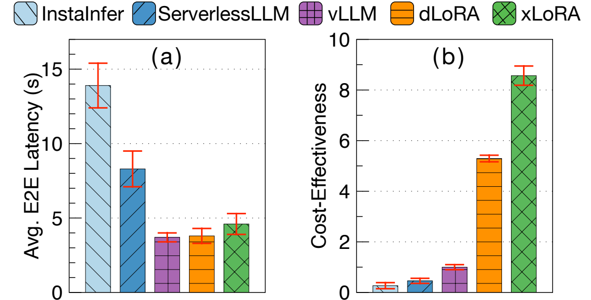


Figure 12: E2E latency and cost-effectiveness for 70B functions on the LLM inference trace.

5.7 Throughput

We evaluate the maximum output token per second and maximum requests completed per second for Llama2-7B series functions. We run these 4 functions concurrently in two GPUs (each GPU has enough capacity for holding two Llama2-7B LLM and their artifacts). As Table 2 shows, *xLoRA* outperforms both ServerlessLLM and InstaInfer, improving the maximum output token throughput 1.65×, the maximum batch size 2.28×, and the throughput up to 3.02×.

The throughput improvement is due to *xLoRA*'s backbone-sharing mechanism. As serving each request requires GPU memory for storing its KV cache, under a large batch size, the KV cache's memory cost is non-negligible. Consequently, as ServerlessLLM and InstaInfer require each function to load the complete backbone LLM, while *xLoRA* only needs to load one backbone LLM in each GPU, functions in *xLoRA* have more GPU memory for holding KV cache. Thus, *xLoRA* can significantly improve throughput under limited GPU resources.

Furthermore, *xLoRA* 's larger peak batch size (compared with other solutions) allows it to serve more concurrent requests—but also introduces greater resource contention. To show whether this contention degrades *xLoRA*'s inference speed, we compare the overall completion times of each solution under the same workload, with all solutions running at their respective maximum batch sizes. Fig. 13 (a) shows that *xLoRA* achieves the shortest completion time. Consequently, even at its peak batch size, when resource contention is highest, *xLoRA* sustains superior inference speed.

5.8 Ablation Study

To evaluate the effectiveness of *xLoRA*'s each component (Backbone Sharing, Pre-Loading, Dynamic Offloading, and

Table 1: The Llama2-7B (13B) series functions' E2E latency, monetary cost, and cost-effectiveness of each solution.

	E2E Latency (ms)			Cost ($)			Cost-Effectiveness (relative)		
Workload	Predictable	Normal	Bursty	Predictable	Normal	Bursty	Predictable	Normal	Bursty
vLLM	2395 (2458)	2425 (2441)	2509 (2573)	20.93 (42.96)	20.93 (42.96)	20.93 (42.96)	1 (1)	1 (1)	1 (1)
dLoRA	2518 (2672)	2589 (2683)	2793 (2856)	7.32 (11.40)	7.32 (11.40)	7.32 (11.40)	2.72 (3.47)	2.68 (3.43)	2.57 (3.39)
InstaInfer	3811 (5777)	4512 (7318)	5620 (7986)	24.32 (53.30)	32.68 (51.46)	24.73 (36.58)	0.55 (0.34)	0.34 (0.34)	0.38 (0.38)
ServerlessLLM	3807 (4733)	4569 (5690)	5168 (6474)	24.29 (43.67)	33.10 (40.01)	22.74 (29.65)	0.55 (0.51)	0.34 (0.51)	0.45 (0.58)
xLoRA	2922 (3166)	3061 (3338)	3050 (3631)	4.66 (7.30)	5.54 (5.87)	3.35 (4.16)	3.71 (4.57)	2.99 (5.35)	5.13 (7.32)

Table 2: Peak throughput of each serverless solution

	Throughput (Token/s)	Peak Batch Size	Throughput (Request/s)
xLoRA	547	73	4.76
ServerlessLLM	331	32	1.72
InstaInfer	331	32	1.48

Table 3: Ablation study of *xLoRA*.

xLoRA Variants	TTFT (ms)	E2E Latency (ms)	Monetary Cost ($)
xLoRA	576	2977	7.53
xLoRA-NBS	2608	4959	12.55
xLoRA-NPL	1345	3747	9.48
xLoRA-NDO	1047	3328	8.42
xLoRA-NAB #1	1386	3799	9.61
xLoRA-NAB #2	693	3425	8.66
xLoRA-NAB #3	721	3526	8.92

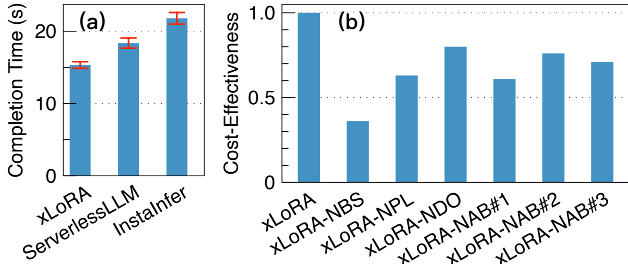


Figure 13: Throughput evaluation and ablation study.

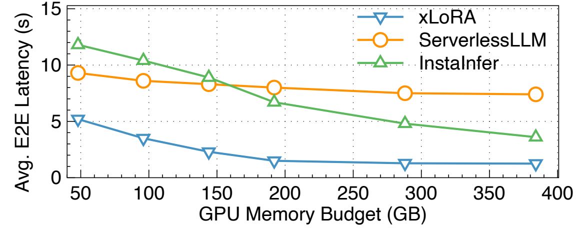


Figure 14: Scalability of *xLoRA* and other solutions.

Adaptive Batching) separately, we conduct an ablation study on the 4-node GPU cluster.

We compare *xLoRA* with its four variants: **1) *xLoRA*-NBS:** *xLoRA* without the Backbone Sharing mechanism. In this variant, each function must independently hold a complete backbone LLM. **2) *xLoRA*-NPL:** *xLoRA* without the Pre-Loading Scheduler. In this variant, no LLM artifacts are pre-loaded. **3) *xLoRA*-NDO:** *xLoRA* without Dynamic Offloading. In this variant, when a bursty workload arrives, if the target GPU does not have enough memory to serve all requests, instead of proactively off-loading unrelated artifacts, it keeps waiting until the GPU has enough memory. **4) *xLoRA*-NAB:** *xLoRA* without the Adaptive Batching Scheduler. In this variant, we set each function's batch size and batch delay to be fixed. For fair comparison, we choose three batching strategies: 1) batch size =1 (no batching); 2) batch size = 10, batch delay = 500 ms; 3) batch size = 20, batch delay = 1000 ms. We name these three strategies *xLoRA*-NAB #1-#3.

We run a 4-hour "Normal" workload of 4 Llama2-7B series functions and 4 Llama2-13B series functions. Fig. 13 (b) shows that *xLoRA* achieves the highest cost-effectiveness among all variants, with *xLoRA*-NBS performing worst, indicating that the backbone sharing mechanism plays the most crucial role in reducing TTFT and monetary cost.

Table 3 details each variant's performance metrics. *xLoRA* achieves the lowest TTFT, E2E latency, and monetary cost. It's worth mentioning that although *xLoRA*-NAB #2 and #3 match *xLoRA*'s TTFT, their fixed batching creates resource contention between different function requests, increasing E2E latency and cost.

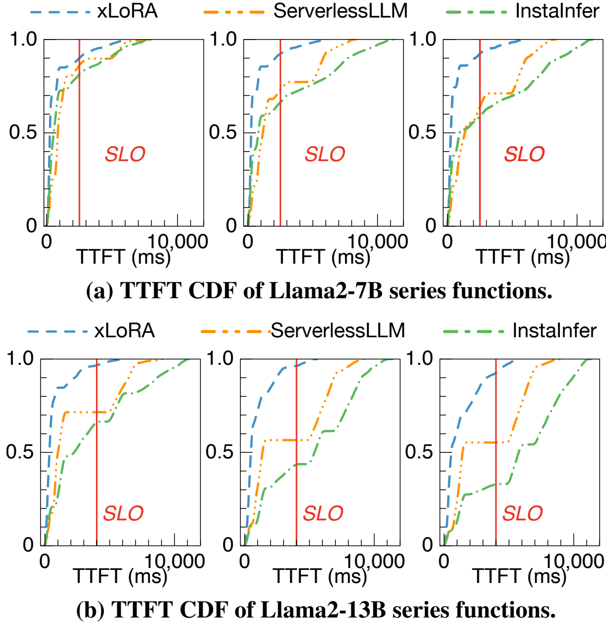
5.9 Scalability

Fig. 14 shows, for the workload that contains all 8 Llama functions, with increasing GPU memory, *xLoRA* consistently outperforms other serverless solutions, indicating its efficient GPU utilization whether GPU resources are limited or abundant. When more GPU memory is available, *xLoRA* effectively converts these resources into faster inference.

5.10 SLO Violation

As *xLoRA* aims to minimize TTFT and monetary cost without violating SLO, we evaluate the SLO violation rate by running the Predictable, Normal, and Bursty workload. We set the TTFT SLO standard following the same setting of ParaServe [33]: 5× the first warm-start's TTFT (without any acceleration of cached kernels). Thus, in our cluster, the TTFT SLO of Llama2-7B series functions is 2500 ms, while that of Llama2-13B series functions is 4000 ms.

Fig. 15 shows that *xLoRA* achieves the lowest SLO violation rate in any workload. Even in the worst case, the violation rate is only 10%, while the SLO violation rate of ServerlessLLM and InstaInfer can reach up to 45% and 58%. This result



(a) TTFT CDF of Llama2-7B series functions.

(b) TTFT CDF of Llama2-13B series functions.

Figure 15: TTFT CDF of *xLoRA* and baselines running the Predictable, Normal, and Bursty workloads.

indicates that *xLoRA*'s outperformance over current serverless solutions does not increase SLO violation rate.

5.11 Prediction Accuracy

We evaluate *xLoRA*'s pre-loading policy with the "Normal" workload trace to assess the impact of prediction accuracy. Our results show a pre-loading hit-rate of at least 47% and up to 82%. Even with a hit-rate lower bound of 12%, *xLoRA* still outperforms InstaInfer and ServerlessLLM.

5.12 Pre-Loading Scheduling

Since artifact pre-loading is an NP-Hard Precedence-Constrained Knapsack Problem, *xLoRA* uses a greedy bin-packing policy for efficiency. Simulations with 50 LoRA adapters, 20 backbone LLMs, 20 idle containers, and 20 GPUs show *xLoRA* achieves near-optimal performance with 260× speedup (0.04s *vs.* 10.4s) at only 1.6% optimality loss.

5.13 Overhead

We measured *xLoRA*'s resource and latency overhead on the above workloads. The Pre-Loading and Adaptive Batching Schedulers each add 1ms latency, with total scheduling overhead under 6ms for the heaviest workload. Backbone sharing introduces no latency overhead. Resource-wise, scheduling components use 1 CPU core and 300MB host memory total. Backbone sharing requires 473MB GPU memory due to separate CUDA contexts for backbone and adapter processes, which is negligible compared to the 14–140GB saving.

6 Related Work

LLM serving: Recent studies focus on accelerating LLM inference and increasing throughput. Orca [60] batches requests at iteration level to minimize queuing time, while FlashAttention [16] reduces IO complexity and SpecInfer [37] uses speculative decoding to reduce latency. DeepSpeed [9] and AlpaServe [31] leverage multi-GPU parallelization. For improved throughput, vLLM [28], InfiniGen [29], and MoonCake [43] optimize KV cache management, while FlexGen [50] and LLM-in-a-flash [6] off-load data to host memory. DistServe [67] and SARATHI [3] disaggregate pre-filling and decoding to maximize GPU utilization. *xLoRA* is orthogonal to these solutions as they optimize inference while *xLoRA* mitigates cold-starts before inference.

Multi-LoRA LLM inference: Multi-LoRA LLMs introduce challenges in efficiently sharing the backbone model without additional memory overhead or degraded inference speed. Punica [11] and S-LoRA [49] support batching requests of different adapters on the same backbone LLM, while dLoRA [58] also accelerates inference by merging LoRA adapters into the backbone. However, these serverful solutions require running models in a single process, violating serverless isolation requirements. *xLoRA* runs each LoRA adapter in independent function instances with isolated adapter models, KV cache, and other data.

Optimizing serverless inference: Serverless ML inference has been well studied [5, 12, 17, 23, 25, 27, 30], but these studies ignore significant model loading delays. Some approaches group requests into batches to improve throughput [4, 59, 64], but cannot flexibly adjust batch size and delay for LLM inference. Works [23, 30, 41] reduce model loading latency, while InstaInfer [53] fully mitigates loading latency of all ML artifacts. ServerlessLLM [20], Medusa [63], ParaServe [33], and λScale [62] accelerate LLM checkpoint loading and compiling. However, all these solutions require maintaining a complete LLM within each function instance, causing high resource costs and limiting scalability.

7 Conclusion

This paper presented *xLoRA*, a serverless inference system designed for efficient LoRA LLM serving. We addressed the backbone redundancy, overlooked artifact loading, and resource contention problems in serverless LoRA inference. *xLoRA* proposes secure backbone LLM sharing, comprehensive LoRA artifact pre-loading, contention-aware adaptive batching, and dynamic GPU memory offloading. Extensive evaluation demonstrates that *xLoRA* significantly reduces TTFT by up to 86% and monetary costs by up to 89% compared to state-of-the-art approaches, offering a faster and more economical solution for deploying multiple LoRA LLMs.

References

[1] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Natisand: Native code sandboxing for javascript runtimes. In *Proc. the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, RAID '23, page 639–653, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

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

[4] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15. IEEE, 2020.

[5] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proc. the VLDB Endowment*, 15(10), 2022.

[6] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. In *Proc. the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 12562–12584, 2024.

[7] Amazon Web Services. Security overview of aws lambda: Lambda isolation technologies. Whitepaper/-Documentation, 2024.

[8] Inc. Amazon Web Services. Amazon bedrock - build generative ai applications with foundation models, 2025. Accessed: 2025-05-15.

[9] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15. IEEE, 2022.

[10] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proc. the Fifteenth European Conference on Computer Systems (EuroSys)*, pages 1–15, 2020.

[11] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *Proc. Machine Learning and Systems (MLSys)*, 6:1–13, 2024.

[12] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proc. Machine Learning and Systems (MLSys)*, 4:20–32, 2022.

[13] Alibaba Cloud. Billable items and billing methods - function compute - alibaba cloud, 2025. Last Updated: Jan 20, 2025. Accessed: May 16, 2025.

[14] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[15] NVIDIA Corporation. Dgx platform: Built for enterprise ai | nvidia, 2025. Accessed: 2025-05-15.

[16] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:16344–16359, 2022.

[17] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proc. the 11th ACM Symposium on Cloud Computing (SoCC)*, pages 45–59, 2020.

[18] Jonatan Enes, Roberto R Expósito, and Juan Touriño. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems*, 105:361–379, 2020.

[19] Hugging Face. Hugging face: Peft llama2 models (sort by most downloads)., 2025. Accessed: 2025-05-08.

[20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Locality-enhanced serverless inference for large language models. *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[21] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proc. the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 386–400, 2021.

[22] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. Fifer: Tackling Underutilization in the Serverless Era. In *The 21st International Middleware Conference (Middleware)*, 2020.

[23] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. Optimus: Warming serverless ml inference via inter-function model transformation. In *Proc. the 19th European Conference on Computer Systems (EuroSys)*, pages 1039–1053, 2024.

[24] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *Proc. the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 455–468, 2019.

[25] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Amps-inf: Automatic model partitioning for serverless inference with cost efficiency. In *Proc. the 50th International Conference on Parallel Processing (ICPP)*, pages 1–12, 2021.

[26] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proc. the 26th ACM international conference on architectural support for programming languages and operating systems (ASPLOS)*, pages 152–166, 2021.

[27] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proc. the 2021 International Conference on Management of Data (SIGMOD)*, pages 857–871, 2021.

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proc. the 29th Symposium on Operating Systems Principles (SOSP)*, pages 611–626, 2023.

[29] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 155–172, 2024.

[30] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[31] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 663–679, 2023.

[32] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing. In *Proc. 2022 USENIX Annual Technical Conference (USENIX ATC)*, pages 69–84, 2022.

[33] Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Xuanzhe Liu, and Xin Jin. Towards swift serverless llm cold starts with paraserve. *arXiv preprint arXiv:2502.15524*, 2025.

[34] HaoHui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. Honeycomb: Secure and efficient GPU executions via static validation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 155–172, Boston, MA, 2023. USENIX Association.

[35] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: Parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

[36] Tuna Han Salih Meral, Enis Simsar, Federico Tombari, and Pinar Yanardag. Clora: A contrastive approach to compose multiple lora models. *arXiv preprint arXiv:2403.19776*, 2024.

[37] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based s peculative inference and verification. In *Proc. The 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 932–949, 2024.

[38] NVIDIA Corporation. *CUDA C++ Best Practices Guide*. NVIDIA Corporation, release 13.0 edition, 2024. Accessed: 2025-09-17.

[39] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. Retention-Aware Container Caching for Serverless Edge Computing. *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2022.

[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132, 2024.

[41] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proc. the ACM Symposium on Cloud Computing (SoCC)*, pages 324–340, 2023.

[42] David Pisinger and Paolo Toth. Knapsack problems. *Handbook of Combinatorial Optimization*, pages 299–428, 1998.

[43] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST)*, pages 155–170, 2025.

[44] Qwen. Qwen chat, 2025. Accessed: 2025-05-15.

[45] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *Proc. 2021 USENIX Annual Technical Conference (USENIX ATC)*, pages 397–411, 2021.

[46] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[47] Mickaël Salaün, Marion Daubignard, and Hervé Debar. Stemjail: Dynamic role compartmentalization. In *Proc. the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*, ASIA CCS '16, page 865–876, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. 2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 205–218, 2020.

[49] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *Proc. Machine Learning and Systems (MLSys)*, 2024.

[50] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning (ICML)*, pages 31094–31116. PMLR, 2023.

[51] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Specfaas: Accelerating serverless applications with speculative function execution. In *Proc. 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–827. IEEE, 2023.

[52] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1348–1362. IEEE, 2025.

[53] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading. In *Proc. the 2024 ACM Symposium on Cloud Computing (SoCC)*, pages 178–195, 2024.

[54] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 495–514, 2021.

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

[56] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. {InfiniCache}: exploiting ephemeral serverless functions to build a {cost-effective} memory cache. In *18th USENIX conference on file and storage technologies (FAST)*, pages 267–281, 2020.

[57] Robert N.M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium (USENIX Security)*, Washington, DC, 2010. USENIX Association.

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

[59] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: A native serverless system for low-latency, high-throughput inference. In *Proc. the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 768–781, 2022.

[60] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 521–538, 2022.

[61] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Libra: Harvesting idle resources safely and timely in serverless clusters. In *Proc. the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 181–194, 2023.

[62] Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, and Ruichuan Chen. λscale: Enabling fast scaling for serverless large language model inference. *arXiv preprint arXiv:2502.09922*, 2025.

[63] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. Medusa: Accelerating serverless llm inference with materialization. In *Proc. the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 653–668, 2025.

[64] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *Proc. 2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 1049–1062, 2019.

[65] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[66] Ming Zhong, Yelong Shen, Shuohang Wang, Yadong Lu, Yizhu Jiao, Siru Ouyang, Donghan Yu, Jiawei Han, and Weizhu Chen. Multi-lora composition for image generation. *arXiv preprint arXiv:2402.16843*, 2024.

[67] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–210, 2024.

[68] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proc. the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1–14, 2022.

A Appendix

A.1 Implementation

We implement a prototype of *xLoRA* with about 5.5K lines of Python code and 600 lines of CUDA code. This prototype includes a complete serverless serving system and all *xLoRA*'s components. We describe the detailed implementation of *xLoRA* as follows:

Backbone LLM sharing. We implement backbone sharing using CUDA IPC handles. After loading the backbone LLM on GPU, we record each layer's CUDA IPC handle. Since Python cannot directly access IPC handles, we create a CUDA plugin compiled into a shared library, enabling Python to access these handles as tensor values. We also modify PyTorch's parameter loading function to populate empty LLM objects by pointing tensors to the backbone LLM with zero-copy operations. Since IPC handles support concurrent access, a single backbone LLM can serve multiple functions simultaneously.

LoRA inference using the shared backbone LLM. As the backbone parameters are shared by multiple functions, regular LoRA inference tools like PEFT [35] that merge the LoRA adapter's parameters with backbone's parameters are unsuitable. Thus, we create the unmerged inference atop Transformers to operate the matrix calculation of backbone and LoRA adapter separately.

LLM artifact pre-loading. We deploy the Pre-Loading Scheduler in the serverless platform's controller, which has access to all worker node information. Each worker node runs a Pre-Loading Agent, and containers include handlers for artifact loading. Components communicate via REST API. To accelerate backbone LLM pre-loading, we use CUDA Streams for concurrent tensor loading and CUDA Asynchronous Memory Transfer to overlap loading with GPU transfers.

Dynamic offloading. We deploy the Dynamic Offloader in each worker node. Once the swapper is triggered, it calls the Pre-Loading Scheduler to trigger the corresponding container's handler for offloading.

A.2 How to use *xLoRA*

To enable *xLoRA*, a minor, two-line modification to the user's code is required, as shown in Code 1.

Code snippet 1: How to use *xLoRA*

```
# Original
model =
    AutoModelForCausalLM.from_pretrained("model")

# xLoRA
import xLoRA
model = xLoRA.from_pretrained("model")
```